Владислав Пирогов

4-е издание

Ассемблер для Windows





- Средства программирования в windows
- Отладка, исследование кода программ, драйверы
- Создание динамических библиотек
- Многозадачное и сетевое программирование
- Графическое программирование GDI+, OpenGL, DirectX
- Работоспособность в ОС Windows Vista

Владислав Пирогов

АССЕМБЛЕР для WINDOWS

4-е ИЗДАНИЕ

Санкт-Петербург

«БХВ-Петербург»

2011

УДК 681.3.068+800.92Assembler
ББК 32.973.26-018.1
П33

Пирогов В. Ю.

П33 Ассемблер для Windows. Изд. 4-е перераб. и доп. — СПб.: БХВ-Петербург, 2011. — 896 с.: ил. + CD-ROM — (Профессиональное программирование)

ISBN 978-8-9775-2184-5

Рассмотрены необходимые сведения для программирования Windows-приложений на ассемблерах MASM и TASM: разработка оконных и консольных приложений; создание динамических библиотек; многозадачное программирование; программирование в локальной сети, в том числе и с использованием сокетов; создание драйверов, работающих в режиме ядра; простые методы исследования программ и др. В 4-м издании материал существенно переработан в соответствии с новыми возможностями ОС. Значительно шире рассмотрены вопросы управления файлами и API-программирования в Windows. Добавлен материал по программированию в ОС семейства Windows NT: Windows 2000/ XP/ Server 2003/Vista. На компакт-диске приведены многочисленные примеры, сопровождающие текст и проверенные на работоспособность в операционной системе Windows Vista.

Для программистов

УДК 681.3.068+800.92Assembler
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.0810.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 72,24.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

Оглавление

Введение 1

Что нового? 2
Соглашения 4
О Windows Vista 5
Структура изложения 5

Введение ко второму изданию книги "Ассемблер для Windows"...........11

Введение к третьему изданию книги "Ассемблер для Windows" 15

ЧАСТЬ I. ОСНОВЫ ПРОГРАММИРОВАНИЯ В WINDOWS 17

Глава 1.1. Средства программирования в Windows 19

Первая программа на языке ассемблера и ее трансляция 19
Объектные модули 24
Директива *INVOKE* 27
Данные в объектном модуле 29
Упрощенный режим сегментации 31
О пакете MASM32..32
Обзор пакета MASM32 33
Трансляторы..35
Редактор QEDITOR 35
Дизассемблеры...37

Глава 1.2. Основы программирования в операционной системе Windows...42

Вызов функций API..44
Структура программы...46
Регистрация класса окон...46
Создание окна 47

Цикл обработки очереди сообщений.....47
Процедура главного окна.....48
Примеры простых программ для Windows.....49
Еще о цикле обработки сообщений.....56
Передача параметров через стек.....58

Глава 1.3. Примеры простых программ на ассемблере.....61

Принципы построения оконных приложений.....61
Окно с кнопкой.....63
Окно с полем редактирования.....68
Окно со списком.....76
Дочерние и собственные окна.....85

Глава 1.4. Ассемблер MASM.....95

Командная строка ML.EXE.....95
Командная строка LINK.EXE.....98
Включение в исполняемый файл отладочной информации.....101
Получение консольных и GUI-приложений.....107
Автоматическая компоновка.....107
"Самотранслирующаяся" программа.....107

Глава 1.5. О кодировании текстовой информации в операционной системе Windows.....109

О кодировании текстовой информации.....109
OEM и ANSI.....110
Кодировка Unicode.....111

ЧАСТЬ II. ПРОСТЫЕ ПРОГРАММЫ, КОНСОЛЬНЫЕ ПРИЛОЖЕНИЯ, ОБРАБОТКА ФАЙЛОВ.....117

Глава 2.1. Вывод графики и текста в окно. Библиотека GDI.....119

Вывод текста в окне.....119
Выбор шрифта.....135
Графические образы.....141

Глава 2.2. Графика: GDI+, DirectX, OpenGL.....154

Работаем с функциями GDI+.....154
Библиотека DirectX.....165
Программируем на OpenGL.....177

Глава 2.3. Консольные приложения ... 191

Создание консоли ... 194
Обработка событий от мыши и клавиатуры ... 200
Событие *KEY_EVENT* ... 201
Событие *MOUSE_EVENT* ... 202
Событие *WINDOW_BUFFER_SIZE_EVENT* ... 203
Таймер в консольном приложении ... 208

Глава 2.4. Понятие ресурса. Редакторы и трансляторы ресурсов ... 217

Язык описания ресурсов ... 217
Пиктограммы ... 218
Курсоры ... 219
Битовые изображения ... 220
Строки ... 220
Диалоговые окна ... 220
Меню ... 226
Акселераторы ... 232
Немодальные диалоговые окна ... 235

Глава 2.5. Примеры программ, использующих ресурсы ... 243

Динамическое меню ... 243
Горячие клавиши ... 254
Управление списками ... 263
Программирование в стиле Windows XP и Windows Vista ... 270

Глава 2.6. Управление файлами: начало ... 277

Характеристики файлов ... 277
Атрибут файла ... 278
Временные характеристики ... 279
Длина файла ... 280
Имя файла ... 281
Файловая система FAT32 ... 282
Файловая система NTFS ... 285
Каталоги в NTFS ... 290
Сжатие файлов в NTFS ... 290
Точки повторной обработки ... 291
Поиск файлов ... 292
Приемы работы с двоичными файлами ... 310
Пример получения временных характеристик файла ... 324

Глава 2.7. Директивы и макросредства ассемблера ... 329

Метки ... 329
Строки ... 332
Структуры ... 332
Объединения ... 333
Удобный прием работы со структурами ... 333
Условное ассемблирование ... 334
Вызов процедур ... 335
Макроповторения ... 336
Макроопределения ... 337
Некоторые другие директивы транслятора ассемблера ... 339
Конструкции времени исполнения программы ... 340
Пример программы одинаково транслируемой как в MASM, так и в TASM ... 342

Глава 2.8. Еще об управлении файлами (*CreateFile* и другие функции) ... 344

Полное описание функции *CreateFile* для работы с файлами ... 344
Другие возможности функции *CreateFile* ... 349
Почтовый ящик или mailslot ... 350
Каналы передачи информации (pipes) ... 355
Дисковые устройства ... 356
Обзор некоторых других функций API, используемых для управления файлами ... 360
Асинхронный ввод/вывод ... 361
Запись в файл дополнительной информации ... 366

ЧАСТЬ III. СЛОЖНЫЕ ПРИМЕРЫ ПРОГРАММИРОВАНИЯ В WINDOWS ... 369

Глава 3.1. Таймер в оконных приложениях ... 371

Общие сведения ... 371
Простейший пример использования таймера ... 373
Взаимодействие таймеров ... 378
Всплывающие подсказки ... 384

Глава 3.2. Многозадачное программирование ... 397

Процессы и потоки ... 397
Потоки ... 412

Взаимодействие потоков 418
Семафоры 420
События 422
Критические секции 422
Взаимоисключения 433

Глава 3.3. Создание динамических библиотек 434

Общие понятия 434
Создание динамических библиотек 437
Неявное связывание 442
Использование общего адресного пространства 443
Совместное использование памяти разными процессами 452

Глава 3.4. Сетевое программирование 456

Сетевые устройства 456
Поиск сетевых устройств и подключение к ним 463
О сетевых протоколах TCP/IP 478
О модели OSI 478
О семействе TCP/IP 479
Об IP-адресации 481
Маскирование адресов 482
Физические адреса и адреса IP 483
О службе DNS 483
Автоматическое назначение IP-адресов 484
Маршрутизация и ее принципы 484
Управление сокетами 485
Пример простейшего клиента и сервера 490

Глава 3.5. Разрешение некоторых проблем программирования в Windows 504

Глава 3.6. Некоторые вопросы системного программирования в Windows 555

Страничная и сегментная адресация 555
Адресное пространство процесса 560
Управление памятью 563
Динамическая память 563
Виртуальная память 571
Фильтры (HOOKS) 572

Глава 3.7. Совместное использование ассемблера с языками высокого уровня 581

Согласование вызовов (исторический экскурс) 581
Согласование имен 582
Согласование параметров 583
Простой пример использования ассемблера с языками высокого уровня 584
Передача параметров через регистры 589
Вызовы API и ресурсы в ассемблерных модулях 591
Развернутый пример использования языков ассемблера и С 597
Встроенный ассемблер 605
Пример использования динамической библиотеки 607
Использование языка С из программ, написанных на языке ассемблера 610

Глава 3.8. Программирование сервисов 615

Основные понятия и функции управления 615
Структура сервисов 618
Пример сервиса 623

ЧАСТЬ IV. ОТЛАДКА, АНАЛИЗ КОДА ПРОГРАММ, ДРАЙВЕРЫ 639

Глава 4.1. Обзор инструментов для отладки и дизассемблирования 641

Утилиты фирмы Microsoft 641
  EDITBIN.EXE 641
  DUMPBIN.EXE 643
  Дизассемблер W32Dasm 646
  Отладчик OllyDbg 646
Другие инструменты 647
  DUMPPE.EXE 647
  Hiew.exe 647
  DEWIN.EXE 650
  IDA Pro 650

Глава 4.2. Отладчик OllyDbg 656

Начало работы с отладчиком 656
  Окна отладчика 656
  Отладочное выполнение 659
Точки останова 660
  Обычные точки останова 660
  Условные точки останова 661
  Условные точки останова с записью в журнал 661

Точка останова на сообщения Windows 661
Точка останова на функции импорта 663
Точка останова на область памяти 663
Точка останова в окне *Memory* 663
Аппаратные точки останова 664
Другие возможности 664
Окно наблюдения 664
Поиск информации 665
Исправление исполняемого модуля 665

Глава 4.3. Описание работы с дизассемблером W32Dasm и отладчиком SoftICE 666

Отладчик W32Dasm 666
Начало работы 666
Передвижение по дизассемблированному тексту 668
Отображение данных 669
Вывод импортированных и экспортированных функций 670
Отображение ресурсов 670
Операции с текстом 671
Загрузка программ для отладки 671
Работа с динамическими библиотеками 673
Точки останова 673
Модификация кода, данных и регистров 673
Поиск нужного места в программе 675
Отладчик SoftICE 676
Основы работы с SoftICE 677
Запуск и интерфейс 677
Краткий справочник по SoftICE 688

Глава 4.4. Основы анализа кода программ 712

Переменные и константы 712
Управляющие структуры языка C 717
Условные конструкции 717
Вложенные условные конструкции 717
Оператор *switch* или оператор выбора 718
Циклы 719
Локальные переменные 720
Функции и процедуры 722
Оптимизация кода 723
Объектное программирование 727

Глава 4.5. Исправление исполняемых модулей ... 732

Простой пример исправления исполняемого модуля ... 732
Пример снятия защиты ... 736
Стадия 1. Попытка зарегистрироваться ... 736
Стадия 2. Избавляемся от надоедливого окна ... 738
Стадия 3. Доводим регистрацию до логического конца ... 740
Стадия 4. Неожиданная развязка ... 741

Глава 4.6. Структура и написание драйверов ... 743

О ядре и структуре памяти ... 743
Управление драйверами ... 745
Пример простейшего драйвера, работающего в режиме ядра ... 747
Драйверы режима ядра и устройства ... 753

ПРИЛОЖЕНИЯ ... 767

Приложение 1. Справочник API-функций и сообщений Windows ... 769

Приложение 2. Справочник по командам и архитектуре микропроцессора Pentium ... 787

Регистры микропроцессора Pentium ... 787
Регистры общего назначения ... 787
Регистр флагов ... 788
Сегментные регистры ... 789
Управляющие регистры ... 789
Системные адресные регистры ... 791
Регистры отладки ... 791
Команды процессора ... 792
Команды арифметического сопроцессора ... 806
Расширение MMX ... 814
О новых инструкциях MMX ... 817

Приложение 3. Защищенный режим микропроцессора Pentium ... 819

Об уровнях привилегий ... 819
Селекторы ... 820
Дескриптор кода и данных ... 820
Другие дескрипторы ... 821
Сегмент TSS ... 822
О защите и уровнях привилегий ... 822

Привилегированные команды 822
Переключение задач 823
Страничное управление памятью 823

Приложение 4. Структура исполняемых модулей 825

Общая структура PE-модуля 826
Заголовок PE-модуля 828
Таблица секций 834
Секция экспорта (*.edata*) 837
Секция импорта (*.idata*) 839
Локальная область данных потоков 841
Секция ресурсов (*.rdata*) 842
Таблица настроек адресов 843
Отладочная информация (*.debug$S*, *.debug$T*) 845

Приложение 5. Файл kern.inc, используемый в главе 4.6 846

Приложение 6. Пример консольного приложения с полной обработкой событий 855

Приложение 7. Описание компакт-диска 865

Список литературы 867

Предметный указатель 869

Введение

Вот уже и до четвертого издания добрались. Путь был нелегкий. Когда я писал первый вариант книги, я не был уверен в ее успехе. Ассемблер в Windows казался экзотической затеей. Но вот появилась книга, и стало приходить большое количество положительных откликов. Я даже не ожидал, что книга будет пользоваться таким успехом. Ко мне приходит большое количество отзывов, на которые, к моему глубокому сожалению, я не всегда, по причине занятости, могу вовремя ответить.

Признаюсь, что иногда я брожу по Интернету и ищу отрицательные отзывы на свои книги. Таких набирается довольно много. К моему глубокому удовлетворению, по книге "Ассемблер для Windows" (см. [22]) я не нашел ни одного отклика, который бы указывал мне на серьезную ошибку в программе или изложении. Встречаются ошибки и погрешности, связанные с моей невнимательностью во время редакторской работы — книга все-таки довольно объемна. В данном издании я постараюсь их исправить. Есть претензии к моему стилю программирования, но я сразу оговорился, что это *мой стиль*, и от него я отступать не намерен. Кроме того, этот стиль продиктован и педагогическими соображениями — учащийся должен видеть все детали, которые часто скрывают при использовании макросредств.

Вы держите в руках новое издание. Чем же продиктовано его появление? Во-первых, мне хотелось избавиться от некоторых устаревших материалов. В первую очередь это касается ассемблера TASM и программирования под Windows 3.1. Во-вторых, я посчитал необходимым расширить рамки книги за счет добавления нового материала по программированию под Windows. И я надеюсь, что читатель здесь не будет разочарован. Наконец, в данной книге я ориентируюсь на операционные системы версии не ниже Windows XP. Кроме Windows XP программы, представленные в книге, были проверены на Windows Server 2003 и Windows Vista. И последнее, к книге на сей раз будет приложен компакт-диск со всеми примерами.

Интернет-поддержку моей книги осуществляет мой сайт **http://asm.shadrinsk.net**. Рад буду встрече там с моими читателями.

Что нового?

Пять лет назад вышло первое издание данной книги. За это время она нашла своих читателей. Некоторые купили все издания данной книги. Ориентируясь как раз на таких поклонников ассемблера, я решил представить некоторый анализ того, что сделано в данном издании по сравнению с предыдущим. Ниже представлена табл. В1, в которой дана сжатая информация о том, как изменились главы предыдущего издания книги в данном издании. В таблице три столбца: крайний левый содержит старое название главы, средний столбец — новое название, крайний правый столбец — краткую информацию того, что произошло с данной главой.

Таблица В1

Старое название главы	**Новое название главы**	**Изменения**
Глава 1.1. Средства программирования в Windows	Глава 1.1. Средства программирования в Windows	Добавлен новый материал
Глава 1.2. Основы программирования в операционной системе Windows	Глава 1.2. Основы программирования в операционной системе Windows	Добавлен новый материал
Глава 1.3. Примеры простых программ на ассемблере	Глава 1.3. Примеры простых программ на ассемблере	Незначительные изменения
Глава 1.4. Экскурс в 16-битное программирование		Глава изъята
Глава 1.5. Ассемблеры MASM и TASM	Глава 1.4. Ассемблер MASM	Значительно переработана
Глава 1.6. О кодировании текстовой информации в операционной системе Windows	Глава 1.5. О кодировании текстовой информации в операционной системе Windows	Незначительные изменения
Глава 2.1. Примеры простейших программ	Глава 2.1. Вывод графики и текста в окно. Библиотека GDI	Незначительные изменения
	Глава 2.2. Графика: GDI+, DirectX, OpenGL	Новая глава

Таблица В1 *(продолжение)*

Старое название главы	**Новое название главы**	**Изменения**
Глава 2.2. Консольные приложения	Глава 2.3. Консольные приложения	Незначительные изменения
Глава 2.3. Понятие ресурса. Редакторы и трансляторы ресурсов	Глава 2.4. Понятие ресурса. Редакторы и трансляторы ресурсов	Незначительные изменения
Глава 2.4. Примеры программ, использующих ресурсы	Глава 2.5. Примеры программ, использующих ресурсы	Незначительные изменения
Глава 2.5. Управление файлами: начало	Глава 2.6. Управление файлами: начало	Незначительные изменения
Глава 2.6. Директивы и макросредства ассемблера	Глава 2.7. Директивы и макросредства ассемблера	Незначительные изменения
Глава 2.7. Еще об управлении файлами (функция *CreateFile* и др.)	Глава 2.8. Еще об управлении файлами (*CreateFile* и другие функции)	Добавлен новый материал
Глава 3.1. Примеры программ, использующих таймер	Глава 3.1. Таймер в оконных приложениях	Незначительные изменения
Глава 3.2. Многозадачное программирование	Глава 3.2. Многозадачное программирование	Значительно переработана
Глава 3.3. Создание динамических библиотек	Глава 3.3. Создание динамических библиотек	Незначительные изменения
Глава 3.4. Программирование в сети	Глава 3.4. Сетевое программирование	Незначительные изменения
Глава 3.5. Разрешение некоторых проблем программирования в Windows	Глава 3.5. Разрешение некоторых проблем программирования в Windows	Значительно переработана. Добавлен новый материал
Глава 3.6. Некоторые вопросы системного программирования в Windows	Глава 3.6. Некоторые вопросы системного программирования в Windows	Добавлен новый материал
Глава 3.7. Использование ассемблера с языками высокого уровня	Глава 3.7. Совместное использование ассемблера с языками высокого уровня	Добавлен новый материал
Глава 3.8. Программирование сервисов	Глава 3.8. Программирование сервисов	Незначительные изменения

Таблица В1 *(окончание)*

Старое название главы	Новое название главы	Изменения
Глава 4.1. Обзор отладчиков и дизассемблеров	Глава 4.1. Обзор инструментов для отладки и дизассемблирования	Значительно переработана
Глава 4.2. Введение в турбоассемблер		Глава изъята
	Глава 4.2. Отладчик OllyDbg	Новая глава
Глава 4.3. Описание работы с дизассемблером W32Dasm и отладчиком ICE	Глава 4.3. Описание работы с дизассемблером W32Dasm и отладчиком SoftICE	Добавлен новый материал
Глава 4.4. Основы анализа кода программ	Глава 4.4. Основы анализа кода программ	Незначительные изменения
Глава 4.5. Исправление исполняемых модулей	Глава 4.5. Исправление исполняемых модулей	Значительно переработана. Добавлен новый материал
Глава 4.6. Структура и написание драйверов	Глава 4.6. Структура и написание драйверов	Удален устаревший материал
Приложения	Приложения	Содержимое значительно переработано. Добавлено приложение с развернутым примером консольной обработки событий

Соглашения

Синонимами в данной книге являются такие термины, как: ассемблер и язык ассемблера; процедура, функция[1], подпрограмма. Под операционной системой Windows в данной книге будут пониматься операционные системы семейства NT — Windows XP, Windows Server 2003, Windows Vista. В более ранних версиях Windows приводимые программы мною не апробировались, хотя большинство из них, скорее всего, будут корректно работать и в других версиях Windows.

[1] Согласитесь, что различия между понятиями "процедура", "функция" и "подпрограмма" могут существовать лишь в языках высокого уровня.

О Windows Vista

Все примеры, которые приведены к книге и размещены, соответственно, на компакт-диске, проверялись в первую очередь на работоспособность в операционной системе Windows Vista, которая, несомненно, довольно скоро станет основной настольной системой от Microsoft. Кроме этого, все скриншоты (за незначительным исключением) окон приложений, представленных в книге, взяты именно из этой операционной системы.

Структура изложения

- *Часть I. Основы программирования в Windows.*
  - *Глава 1.1. Средства программирования в Windows.*

    Первая программа на языке ассемблера и ее трансляция. Объектные модули. Директива `INVOKE`. Данные в объектном модуле. Упрощенный режим сегментации. Пакет MASM32 — обзор утилит, возможностей и библиотек. Дается краткое описание средств программирования на ассемблере: трансляторов, компоновщиков, отладчиков и т. п.
  - *Глава 1.2. Основы программирования в операционной системе Windows.*

    Вызов функций API. Структура программы. Примеры простых программ для Windows. Передача параметров через стек. Описываются основные структуры на языке ассемблера.
  - *Глава 1.3. Примеры простых программ на ассемблере.*

    Принципы построения оконных приложений. Приводятся примеры программ для Windows с их подробными комментариями. Окно с кнопкой. Окно с полем редактирования. Окно со списком. Дочерние и собственные окна.
  - *Глава 1.4. Ассемблер MASM.*

    Командные строки LINK.EXE и ML.EXE. Получение консольных и GUI-приложений. Автоматическая компоновка. Использование пакетных файлов. "Самотранслирующаяся" программа.
  - *Глава 1.5. О кодировании текстовой информации в операционной системе Windows.*

    О кодировании текстовой информации. OEM и ANSI. Кодировка Unicode. API-функции, полезные при работе с текстовой информацией. Примеры перекодировок.

- *Часть II. Простые программы, консольные приложения, обработка файлов.*
  - *Глава 2.1. Вывод графики и текста в окно. Библиотека GDI.*

    Приводятся примеры простейших 32-битных программ с выводом в окно и с подробными пояснениями. Вывод текста в окне. Выбор шрифта. Графические образы. Библиотека GDI.
  - *Глава 2.2. Графика: GDI+, DirectX, OpenGL.*

    Обзор графических библиотек Windows: GDI+, DirectX, OpenGL с подробными примерами.
  - *Глава 2.3. Консольные приложения.*

    Понятие консольного приложения. Создание консоли. Обработка событий от мыши и клавиатуры. Таймер в консольном приложении. Общая событийная модель консольного приложения.
  - *Глава 2.4. Понятие ресурса. Редакторы и трансляторы ресурсов.*

    Язык описания ресурсов. Редакторы ресурсов. Трансляторы ресурсов. Немодальные диалоговые окна как элементы ресурсов.
  - *Глава 2.5. Примеры программ, использующих ресурсы.*

    Продолжение работы с ресурсами. Динамическое меню. Горячие клавиши. Управление списками. Программирование в стиле Windows XP.
  - *Глава 2.6. Управление файлами: начало.*

    Излагаются основы файловых систем Windows FAT32 и NTFS. Характеристики файлов. Файловая система FAT32. Файловая система NTFS. Поиск файлов. Приемы работы с двоичными файлами. Пример получения временных характеристик файла. Дается описание основных API-функций работы с файлами, приводятся примеры программ с файловой обработкой, пример рекурсивного поиска файлов по дереву каталогов.
  - *Глава 2.7. Директивы и макросредства ассемблера.*

    Макровозможности MASM32. Метки. Структуры. Объединения. Удобный прием работы со структурами. Условное ассемблирование. Вызов процедур. Макроповторения. Макроопределения. Некоторые другие директивы транслятора ассемблера. Конструкции времени исполнения программы. Принцип разработки программ, транслируемых различными ассемблерами.
  - *Глава 2.8. Еще об управлении файлами (CreateFile и другие функции).*

    Более углубленное изучение файлов. Полное описание функции `CreateFile` для работы с файлами. Другие возможности функции

`CreateFile`. Обзор некоторых других функций API, используемых для управления файлами. Асинхронный ввод/вывод. Запись в поток файла.

- *Часть III. Сложные примеры программирования в Windows.*
  - *Глава 3.1. Таймер в оконных приложениях.*

    Таймеры. Простейший пример использования таймера. Взаимодействие таймеров. Теория всплывающих подсказок.
  - *Глава 3.2. Многозадачное программирование.*

    Многозадачность. Процессы и потоки. Создание процесса. Потоки. Взаимодействие потоков. Семафоры. События. Критические секции. Взаимоисключения. Примеры создания и управления потоками.
  - *Глава 3.3. Создание динамических библиотек.*

    Общие понятия. Динамические библиотеки, их структура. Создание динамических библиотек. Неявное связывание. Использование общего адресного пространства. Совместное использование памяти разными процессами.
  - *Глава 3.4. Сетевое программирование.*

    Элементы сетевого программирования. Сетевые устройства. Поиск сетевых устройств и подключение к ним. О сетевых протоколах TCP/IP. Управление сокетами. Примеры простейшего клиента и сервера.
  - *Глава 3.5. Разрешение некоторых проблем программирования в Windows.*

    Примеры интересных задач. Значок на системной панели инструментов. Средства файловой обработки. Контроль данных в окне ввода. Обмен данными между приложениями. Предотвращение многократного запуска приложения. Операции над группами файлов или каталогов. Использование списка задач и списка окон.
  - *Глава 3.6. Некоторые вопросы системного программирования в Windows.*

    О защищенном режиме. О страничной и сегментной адресации. Адресное пространство процесса. Управление памятью: динамическая и виртуальная память. Управление памятью. Файлы, проецируемые в память. Фильтры (Hooks). Перехват функций API.
  - *Глава 3.7. Использование ассемблера с языками высокого уровня.*

    Ассемблер и языки высокого уровня (ЯВУ). Согласование вызовов. Согласование имен. Согласование параметров. Простой пример использования ассемблера с языками высокого уровня. Передача параметров

через регистры. Вызовы API и ресурсы в ассемблерных модулях. Развернутый пример совместного использования языков ассемблера и С. Использование языка С и библиотек языка С из языка ассемблера. Встроенный ассемблер. Пример использования динамической библиотеки на языке высокого уровня.

- *Глава 3.8. Программирование сервисов.*

  Сервисы. Понятие сервиса. Основные понятия и функции управления. Структура сервисов. Пример программирования сервиса.

❒ *Часть IV. Отладка, анализ кода программ, драйверы.*

- *Глава 4.1. Обзор инструментов для отладки и дизассемблирования.*

  Утилиты, используемые при отладке и дизассемблировании исполняемого кода: dumpbin.exe, hiew.exe и др.

- *Глава 4.2. Отладчик OllyDbg.*

  Отладчик OllyDbg. Окна отладчика. Выполнение под отладчиком. Точки останова. Исправление исполняемого кода и др.

- *Глава 4.3. Описание работы с дизассемблером W32Dasm и отладчиком SoftICE.*

  Дизассемблер и отладчик W32Dasm. Интерфейс и настройка программы. Работа с дизассемблируемым кодом. Отладка программ. Отладчик уровня ядра SoftICE. Инсталляция, загрузка, особенности отладки. Справочник команд отладчика SoftICE.

- *Глава 4.4. Основы анализа кода программ.*

  Некоторые парадигмы исследования исполняемого кода. Переменные и константы. Управляющие структуры языка С. Локальные переменные. Функции и процедуры. Оптимизация кода. Объектное программирование.

- *Глава 4.5. Исправление исполняемых модулей.*

  Примеры исследования и исправления исполняемых модулей.

- *Глава 4.6. Структура и написание драйверов.*

  Драйверы, работающие в режиме ядра. Основные понятия. Пример простейшего драйвера, работающего в режиме ядра. Драйверы режима ядра и устройства.

❒ *Приложения.*

- *Приложение 1. Справочник API-функций и сообщений Windows.*

  Приложение содержит краткое описание API-функций и сообщений Windows, которые упоминаются в книге.

- *Приложение 2. Справочник по командам и архитектуре микропроцессора Pentium.*

  Дан полный справочник команд микропроцессора Pentium, кратко описана его архитектура.

- *Приложение 3. Защищенный режим микропроцессора Pentium.*

  Дано описание защищенного режима микропроцессора Intel Pentium.

- *Приложение 4. Структура исполняемых модулей.*

  Дано описание структуры исполняемых модулей операционной системы Windows.

- *Приложение 5. Файл kern.inc, используемый в главе 4.6.*

  В приложении содержится файл kern.inc, используемый в *главе 4.6* при написании драйверов режима ядра.

- *Приложение 6. Пример консольного приложения с полной обработкой событий.*

  Представлен полный пример консольного приложения, обрабатывающего все события от клавиатуры и мыши.

- *Приложение 7. Описание компакт-диска.*

  Дано описание компакт-диска, прилагаемого к книге.

Введение ко второму изданию книги "Ассемблер для Windows"

Если Вы, дорогой читатель, знакомы с книгой "Assembler: учебный курс" Вашего покорного слуги, то, наверное, обратили внимание, что программированию в операционной системе Windows было посвящено всего две главы. Это немного и может служить лишь введением в данную область. Пришло время заняться этим серьезно.

Прежде всего, как и полагается во введении, отвечу на возможное замечание: зачем нужен ассемблер в Windows, если есть, например, Си и другие языки? Зачем нужен ассемблер, я уже писал в упомянутой выше книге. Позволю себе процитировать ее: "Зачем нужен язык ассемблера? — спросят меня. Самой простой и убедительный ответ на поставленный вопрос такой — затем, что это язык процессора и, следовательно, он будет нужен до тех пор, пока будут существовать процессоры. Более пространный ответ на данный вопрос содержал бы в себе рассуждение о том, что ассемблер может понадобиться для оптимизации кода программ, написания драйверов, трансляторов, программирования некоторых внешних устройств и т. д. Для себя я, однако, имею и другой ответ: программирование на ассемблере дает ощущение власти над компьютером, а жажда власти — один из сильнейших инстинктов человека".

Что касается операционной системы Windows[1], то здесь, как ни странно это прозвучит для уха некоторых программистов, программировать на ассемблере гораздо легче, чем в операционной системе MS-DOS. В данной книге я берусь доказать, что программировать на ассемблере в Windows ничуть не сложнее, чем на Си, и при этом получается компактный, эффективный и быстрый код. Работая с языками высокого уровня, мы теряем определенные

---

[1] Под термином "операционная система Windows" в данной книге я подразумеваю сразу несколько по сути разных операционных систем: Windows 95/98, Windows NT, Windows 2000 *(см. далее)*. При необходимости мы будем оговаривать, какую ОС имеем в виду.

алгоритмические навыки. И процесс заходит все дальше. Честное слово, только ради повышения своего профессионального уровня стоит заниматься программированием на ассемблере.

Как и предыдущая, эта книга будет содержать только работающие программы с подробным разбором и комментарием.

В настоящее время наиболее часто используются два ассемблера: MASM (Microsoft Assembler) и TASM (Turbo Assembler). Именно на них мы сконцентрируем наше внимание в книге.

И еще, в книгу вошел материал, который можно назвать "хакерским". Мы рассмотрим способы и средства анализа и исправления кода программ. Для тех, кто начнет говорить о безнравственности исправления чужих программ, замечу, что хакеры все равно существуют, а раз так, то почему бы и не познакомиться с тем, как они работают. Это будет полезно многим программистам.

Надо сказать, что в литературе по программированию для Windows 9*x* образовалась некоторая брешь — авторы очень быстро перешли от чистого API-программирования[2] к описанию визуальных компонентов тех или иных языков. Автору известна лишь одна, да и то переводная, книга по "чистому" программированию для Windows — Герберт Шилдт "Программирование на С и С++ для Windows 95" (см. [4]). В своей книге я пытаюсь прикрыть эту брешь, рассматривая некоторые мало освещенные в литературе вопросы: программирование в локальной сети, использование многозадачности, написание V*x*D-драйверов, обработка файлов и др.

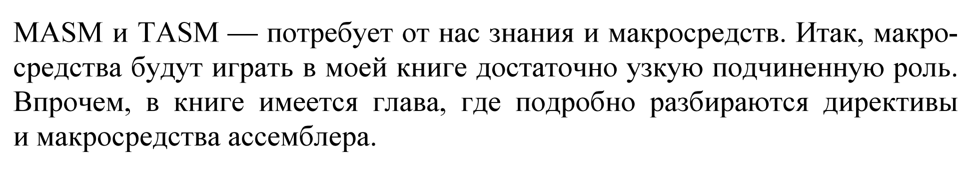
Обычно книги по программированию тяготеют к одной из двух крайностей: описание языка программирования, описание возможностей операционной системы. Мне хотелось удержаться посередине. Данная книга — не руководство по языку ассемблера и не руководство по программированию в Windows. Это нечто среднее, можно сказать — симбиоз языка ассемблера и операционной системы Windows. Как я справился с данной задачей — судить Вам, дорогой читатель.

Изложу некоторые принципы, на которые я опирался, когда писал данную книгу.

- Детальное изложение рассматриваемых вопросов. Я не очень жалую макросредства[3] и считаю, что начинающим программистам их не стоит использовать. Однако цель, которую я преследую в книге: сблизить позиции

[2] Под API-программированием мы понимаем программирование с использованием одних API-функций.

[3] См. книгу автора "Assembler: учебный курс" — [1].



MASM и TASM — потребует от нас знания и макросредств. Итак, макросредства будут играть в моей книге достаточно узкую подчиненную роль. Впрочем, в книге имеется глава, где подробно разбираются директивы и макросредства ассемблера.

- Чтобы сделать изложение максимально полезным, все программы будут излагаться либо в двух вариантах — для MASM и для TASM, либо с подробным объяснением того, как перейти к другому ассемблеру. За основу взят пакет MASM версии 7.0 и TASM (TASM32.EXE версии 5.0, TLINK32.EXE версии 1.6.71). Читателям рекомендую пользоваться версиями не ниже указанных.
- Книга содержит в себе изложение материала, начиная с простых программ и заканчивая элементами системного программирования. Поэтому книгу можно считать специальным учебным курсом по программированию в операционной системе Windows. Желательно (хотя не обязательно) знакомство читателя с языком Си и весьма желательно наличие начальных знаний по языку ассемблера. В качестве учебников по языку ассемблера можно рекомендовать книги [1, 13].
- Книга содержит обширный справочный материал для того, чтобы читатель не отвлекался на поиски его в других книгах и в Интернете. В *приложении 2* имеется справочник по командам микропроцессора с пояснениями. Более подробное объяснение команд можно найти в книгах [1, 3, 13].
- Хорошее знание ассемблера помогает легко разбираться в коде программ. Взломщики чужих программ всегда хорошо владеют ассемблером. Вопросы анализа кода программ не часто рассматривают в компьютерной литературе. Знание этого не только не помешает любому программисту, но и поможет ему защищать свои программы более эффективно.

Введение к третьему изданию книги "Ассемблер для Windows"

Некоторое время назад вышла моя книга "Ассемблер для Windows". Неожиданно для меня она имела довольно высокий рейтинг продаж. По отзывам, которые я получаю, оказалось, что такую книгу ждали. Дело в том, что долгое время писать на ассемблере означало писать под DOS. Выход на арену операционной системы Windows 95 нанес чувствительный удар по программированию на ассемблере. В определенном смысле ассемблер не оправился от этого удара до сих пор. А ведь уже во всю работают операционные системы Windows XP и Windows Server 2003. Своей работой мне хотелось бы вернуть ассемблеру его несколько пошатнувшиеся позиции.

Данная книга строится на основе уже упомянутой мною книги "Ассемблер для Windows". Значительная часть материала взята именно оттуда. Однако, во-первых, этот материал подвергся определенной переработке и уточнению. Уточнения относятся как к самому языку ассемблера, так и к новым возможностям операционных систем. Во-вторых, в книгу добавлен новый материал, касающийся возможностей операционных систем семейства Windows NT, к коим я отношу Windows 2000[1], Windows XP и Windows Server 2003. Например, появилась глава, посвященная сервисам, рассматривается создание драйверов, работающих в режиме ядра и др. Мною добавлен материал и по вопросам, которые были изложены в предыдущей книге. Например, более чем в два раза увеличился объем страниц, посвященных управлению файлами. Вообще значительно расширен материал, посвященный вопросам API-программирования в Windows.

Отмечу также, что все примеры, приведенные в книге, в том числе и перешедшие из книги "Ассемблер для Windows" и, возможно, несколько переработанные, проверялись мною на этот раз именно в операционных системах семейства NT, и, следовательно, я не могу гарантировать их корректную работу в операционных системах семейства Windows 9*x*/Windows ME. То же я

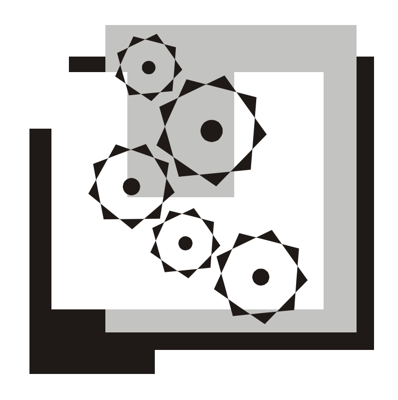
[1] Windows 2000 имеет и другое название: Windows NT 5.0.

могу сказать и о процессоре: все примеры проверялись на процессорах Pentium III и Pentium 4.

Как и в предыдущей книге, я использую два ассемблера — MASM и TASM. Не так давно TASM был продан фирмой Borland компании Paradigm и развивается теперь под другим названием (PASM). Поскольку, однако, среди программистов, пишущих на ассемблере, TASM остается довольно популярным средством, я по-прежнему строю свое изложение, опираясь (там, где это возможно) на оба компилятора.

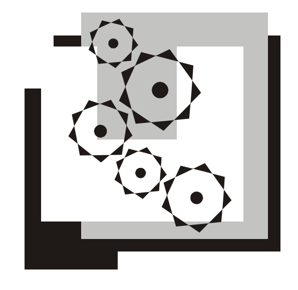
Данная книга в значительной степени отражает пристрастия автора в области программирования и преподавания. В первую очередь, это коснулось макросредств ассемблера. Мне кажется, они (макросредства) в значительной степени скрывают от нас красоту и возможности ассемблера. Я полагаю, что, говоря о программировании как о технологии и в этой связи о стиле программирования, мы забываем, что для многих программирование — это еще и средство самовыражения. Эти две стороны программирования часто входят в противоречие друг с другом, но это уже особый, я бы сказал, философский разговор, и в данной книге мы заниматься этим не будем.

Несколько слов скажу о нумерации глав в данной книге. Книга разбита на части, а те, в свою очередь, на главы. В каждой части своя нумерация глав. Полный номер главы состоит из номера части и номера главы в ней. Таким образом, *глава 2.3* означает главу 3 из части II. Номер рисунка содержит в себе номер части, номер главы и номер рисунка в главе. Программы и фрагменты программ называются листингами, и принцип их нумерации такой же, как у рисунков.



Часть I

ОСНОВЫ ПРОГРАММИРОВАНИЯ В WINDOWS



Глава 1.1

Средства программирования в Windows

Начнем, дорогой читатель. В данной главе я намерен дать некоторую вводную информацию о средствах программирования на языке ассемблера, а также предоставить начальные сведения о трансляции с языка ассемблера. Эта глава предназначена для начинающих программировать на языке ассемблера, поэтому программистам более опытным ее можно пропустить без особого ущерба для себя. Кроме этого, я собираюсь обзорно остановиться на возможностях пакета MASM32. Вместе с тем, начинающим я бы рекомендовал книгу [26], где программирование на языке ассемблера для операционной системы Windows излагается более последовательно и систематически.

Первая программа на языке ассемблера и ее трансляция

Рассмотрим общую схему трансляции программы, написанной на языке ассемблера. В исходном состоянии мы имеем один или более модулей на языке ассемблера (рис. 1.1.1). Двум стадиям трансляции с языка ассемблера соответствуют две программы пакета MASM32: ассемблер ML.EXE и редактор связей LINK.EXE[1]. Эти программы предназначены для того, чтобы перевести модуль (или модули) на языке ассемблера в исполняемый модуль, состоящий из команд процессора и имеющий расширение exe в операционной системе Windows.

Пусть файл с текстом программы на языке ассемблера называется PROG.ASM. Тогда, не вдаваясь в подробный анализ, две стадии трансляции на языке команд будут выглядеть следующим образом:

```
c:\masm32\bin\ml /c /coff PROG.ASM
```

---

[1] Программу LINK.EXE называют также компоновщиком или просто линковщиком.

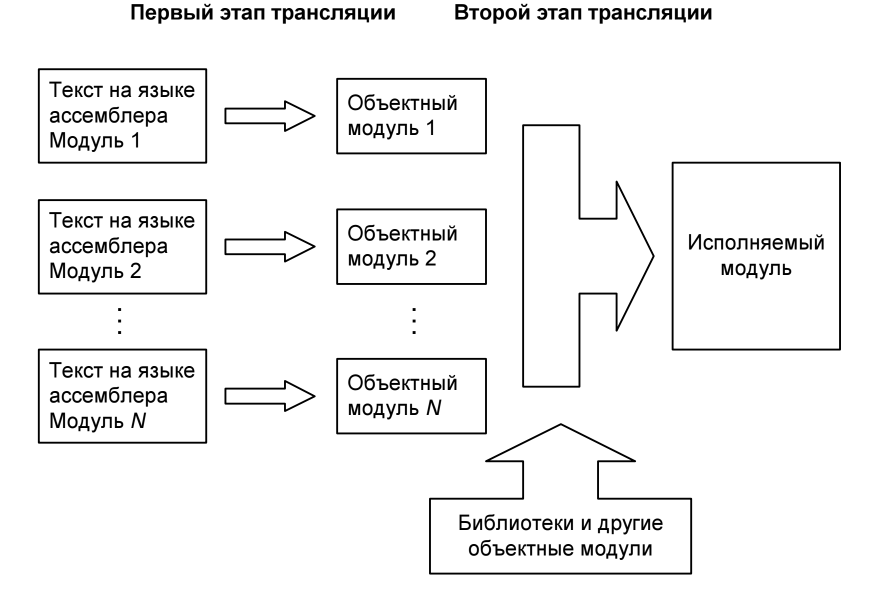


Рис. 1.1.1. Схема трансляции ассемблерного модуля

После работы данной команды появляется *объектный модуль* (см. рис. 1.1.1) PROG.OBJ

```
c:\masm32\bin\Link /SUBSYSTEM:WINDOWS PROG.OBJ
```

В результате последней команды появляется исполняемый модуль PROG.EXE. Как вы, я надеюсь, догадались, `/c` и `/coff` являются параметрами программы ML.EXE, а `/SUBSYSTEM:WINDOWS` — параметром для программы LINK.EXE. О других ключах этих программ более подробно я расскажу в *главе 1.4*.

Чем больше я размышляю об этой схеме трансляции, тем более совершенной она мне кажется. Действительно, формат конечного модуля зависит от операционной системы. Установив стандарт на структуру объектного модуля, мы получаем возможность:

- использовать уже готовые объектные модули,
- интегрировать программы, написанные на разных языках *(см. главу 3.7)*.

Но самое прекрасное здесь то, что если стандарт объектного модуля распространить на разные операционные системы, то можно использовать модули, написанные в разных операционных системах[2].

---

[2] Правда, весьма ограниченно, т. к. согласование системных вызовов в разных операционных системах может вызвать весьма сильные затруднения.

Чтобы процесс трансляции сделать для вас привычным, рассмотрим несколько простых, "ничего не делающих" программ. Первая из них представлена в листинге 1.1.1.

Листинг 1.1.1. "Ничего не делающая" программа

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;----------------------------------------------------
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
      RET   ;выход
_TEXT ENDS
END START
```

В листинге 1.1.1 представлена "ничего не делающая" программа. Назовем ее PROG1. Сразу отмечу на будущее: команды микропроцессора и директивы макроассемблера будем писать заглавными буквами.

Итак, чтобы получить загружаемый модуль, выполним следующие команды:[3]

```
ml /c /coff PROG1.ASM
link /SUBSYSTEM:WINDOWS PROG1.OBJ
```

Примем пока параметры трансляции программ как некую данность и продолжим наши изыскания.

Часто удобно разбить текст программы на несколько частей и объединить эти части еще на 1-й стадии трансляции. Это достигается посредством директивы INCLUDE. Например, один файл будет содержать код программы, а константы, данные (определение переменных) и прототипы внешних процедур помещаются в отдельные файлы. Обычно такие файлы записывают с расширением inc.

Именно такая разбивка демонстрируется в следующей программе (листинг 1.1.2).

[3] Если имя транслируемых модулей содержит пробелы, то название модулей придется заключать в кавычки, например, так: `ML /c /coff "PROG 1.ASM"`.

Листинг 1.1.2. Пример использования директивы INCLUDE

```
;файл CONSTANTS.INC
CONS1 EQU 1000
CONS2 EQU 2000
CONS3 EQU 3000
CONS4 EQU 4000
CONS5 EQU 5000
CONS6 EQU 6000
CONS7 EQU 7000
CONS8 EQU 8000
CONS9 EQU 9000
CONS10 EQU 10000
CONS11 EQU 11000
CONS12 EQU 12000
;файл DATA.INC
DAT1 DWORD 0
DAT2 DWORD 0
DAT3 DWORD 0
DAT4 DWORD 0
DAT5 DWORD 0
DAT6 DWORD 0
DAT7 DWORD 0
DAT8 DWORD 0
DAT9 DWORD 0
DAT10 DWORD 0
DAT11 DWORD 0
DAT12 DWORD 0

;файл PROG1.ASM
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;подключить файл констант
INCLUDE CONSTANTS.INC
;-----------------------------------------------------------
;сегмент данных
_DATA SEGMENT
;подключить файл данных
INCLUDE DATA.INC
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
       MOV EAX,CONS1
       SHL EAX,1  ;умножение на 2
```

```
        MOV DAT1,EAX
;-----------------------------------
        MOV EAX,CONS2
        SHL EAX,2  ;умножение на 4
        MOV DAT2,EAX
;-----------------------------------
        MOV EAX,CONS3
        ADD EAX,1000 ;прибавим 1000
        MOV DAT3,EAX
;-----------------------------------
        MOV EAX,CONS4
        ADD EAX,2000 ;прибавим 2000
        MOV DAT4,EAX
;-----------------------------------
        MOV EAX,CONS5
        SUB EAX,3000 ;вычесть 3000
        MOV DAT5,EAX
;-----------------------------------
        MOV EAX,CONS6
        SUB EAX,4000 ;вычесть 4000
        MOV DAT6,EAX
;-----------------------------------
        MOV EAX,CONS7
        MOV EDX,3
        IMUL EDX ;умножение на 3
        MOV DAT7,EAX
;-----------------------------------
        MOV EAX,CONS8
        MOV EDX,7 ;умножение на 7
        IMUL EDX
        MOV DAT8,EAX
;-----------------------------------
        MOV EAX,CONS9
        MOV EBX,3 ;деление на 3
        MOV EDX,0
        IDIV EBX
        MOV DAT9,EAX
;-----------------------------------
        MOV EAX,CONS10
        MOV EBX,7 ;деление на 7
        MOV EDX,0
        IDIV EBX
        MOV DAT10,EAX
;-----------------------------------
        MOV EAX,CONS11
        SHR EAX,1 ;деление на 2
```

```
        MOV DAT11,EAX
;----------------------------------
        MOV EAX,CONS12
        SHR EAX,2 ;деление на 4
        MOV DAT12,EAX
;----------------------------------
        RET           ;выход
_TEXT ENDS
END START
```

Трансляция программы из листинга 1.1.2:

```
ml /c /coff prog1.asm
link /subsystem:windows prog1.obj
```

Программа из листинга 1.1.2 также достаточно бессмысленна (как и все программы данной главы), но зато демонстрирует удобства использования директивы `INCLUDE`. Напомню, что мы не останавливаемся в книге на очевидных командах микропроцессора *(см. приложение 2)*. Замечу только по поводу команды `IDIV`. В данном случае команда `IDIV` осуществляет операцию деления над операндом, находящемся в паре регистров `EDX:EAX`. Обнуляя `EDX`, мы указываем, что операнд целиком находится в регистре `EAX`.

Трансляция программы осуществляется так, как это было указано ранее.

> ***Замечание о типах данных***
>
> В данной книге вы встретитесь в основном с тремя типами данных (простых): байт, слово, двойное слово. При этом используются следующие стандартные обозначения. Байт — `BYTE` или `DB`, слово — `WORD` или `DW`, двойное слово — `DWORD` или `DD`. Выбор, скажем, в одном случае `DB`, а в другом `BYTE`, продиктован лишь желанием автора несколько разнообразить изложение. Подробнее *см. в главе 2.7*.

Объектные модули

Перейдем теперь к вопросу об объединении нескольких объектных модулей и подсоединении объектных библиотек на второй стадии трансляции. Прежде всего, замечу, что, сколько бы ни объединялось объектных модулей, один объектный модуль является главным. Смысл этого весьма прост: именно с этого модуля начинается исполнение программы. На этом различие между модулями заканчивается. Условимся далее, что главный модуль всегда в начале сегмента кода будет содержать метку `START`, ее мы указываем после директивы `END` — транслятор должен знать точку входа программы, чтобы указать ее в заголовке загружаемого модуля *(см. приложение 4)*.

Обычно во второстепенные модули помещаются процедуры, которые будут вызываться из основного и других модулей. Рассмотрим пример такого модуля. Этот модуль вы можете видеть в листинге 1.1.3.

Листинг 1.1.3. Модуль PROG2.ASM, процедура которого `PROC1` будет вызываться из основного модуля

```
.586P
;модуль PROG2.ASM
;плоская модель памяти
.MODEL FLAT, STDCALL
PUBLIC PROC1
_TEXT SEGMENT
PROC1 PROC
      MOV EAX,1000
      RET
PROC1 ENDP
_TEXT ENDS
END
```

Прежде всего, обращаю ваше внимание на то, что после директивы `END` не указана какая-либо метка. Ясно, что это не главный модуль, процедуры его будут вызываться из других модулей. Другими словами, все его точки входа вторичны и совпадают с адресами процедур, которые там расположены.

Второе, на что я хотел бы обратить ваше внимание, — это то, что процедура, которая будет вызываться из другого объектного модуля, должна быть объявлена как `PUBLIC`. Тогда это имя будет сохранено в объектном модуле и далее может быть связано с вызовами из других модулей программой LINK.EXE.

Итак, выполняем команду `ML /coff /c PROG1.ASM`. В результате на диске появляется объектный модуль PROG2.OBJ.

А теперь проведем маленькое исследование. Просмотрим объектный модуль с помощью какой-нибудь простой viewer-программы, например той, что есть у программы Far.exe. И что же мы обнаружим? Вместо имени `PROC1` мы увидим имя `_PROC1@0`. Это особый разговор — будьте сейчас внимательны! Во-первых, подчеркивание в начале имени отражает стандарт ANSI, предписывающий всем внешним именам (доступным нескольким модулям) автоматически добавлять символ подчеркивания.[4] Здесь ассемблер будет действовать автоматически, и у нас по этому поводу не будет никаких забот.

---

[4] В дальнейшем *(см. главу 3.7)* мы узнаем, что иногда требуется, чтобы символ подчеркивания отсутствовал в объектном модуле.

Сложнее с припиской @0. Что она значит? На самом деле все просто: цифра после знака @ указывает количество байтов, которые необходимо передать в стек в виде параметров при вызове процедуры. В данном случае ассемблер понял так, что наша процедура параметров не требует. Сделано это для удобства использования директивы INVOKE. Но о ней речь пойдет далее, а пока попытаемся сконструировать основной модуль PROG1.ASM (листинг 1.1.4).

Листинг 1.1.4. Модуль PROG1.ASM с вызовом процедуры из модуля PROG2.ASM

```
.586P
; плоская модель памяти
.MODEL FLAT, STDCALL
;--------------------------------------------------------------
;прототип внешней процедуры
EXTERN   PROC1@0:NEAR
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
CALL PROC1@0
RET   ;выход
_TEXT ENDS
END START
```

Как вы понимаете, процедура, которая расположена в другом модуле, но будет вызываться из данного модуля, объявляется как EXTERN. Далее, вместо имени PROC1 нам приходится использовать имя PROC1@0. Здесь пока ничего нельзя сделать. Может возникнуть вопрос о типе NEAR. Дело в том, что когда-то в операционной системе MS-DOS тип NEAR означал, что вызов процедуры (или безусловный переход) будет происходить в пределах одного сегмента. Тип FAR означал, что процедура (или переход) будет вызываться из другого сегмента. В операционной системе Windows реализована так называемая *плоская модель памяти*, когда все адресное пространство процесса можно рассматривать как один большой сегмент. И здесь логично использование типа NEAR.

Выполним команду ML /coff /c PROG1.ASM, в результате получим объектный модуль PROG1.OBJ. Теперь можно объединить модули и получить загружаемую программу PROG1.EXE:

```
LINK /SUBSYSTEM:WINDOWS PROG1.OBJ PROG2.OBJ
```

При объединении нескольких модулей первым должен указываться главный, а остальные — в произвольном порядке. Название исполняемого модуля тогда будет совпадать с именем главного модуля.

Директива *INVOKE*

Обратимся теперь к директиве `INVOKE`. Довольно удобная команда, я вам скажу, правда, по некоторым причинам (которые станут понятными позже) я почти не буду употреблять ее в своих программах.

Удобство ее заключается, во-первых, в том, что мы сможем забыть о добавке `@N`. Во-вторых, эта команда сама заботится о помещении передаваемых параметров в стек. Последовательность команд

```
PUSH par1
PUSH par2
PUSH par3
PUSH par4
CALL NAME_PROC@N ; N - количество отправляемых в стек байтов
```

заменяется на

```
INVOKE NAME_PROC, par4, par3, par2, par1
```

Причем параметрами могут являться регистр, непосредственно значение или адрес. Кроме того, для адреса может использоваться как оператор `OFFSET`, так и оператор `ADDR` *(см. главу 2.6)*.

Видоизменим теперь модуль PROG1.ASM (модуль PROG2.ASM, представленный в листинге 1.1.3, изменять не придется). Измененный модуль расположен в листинге 1.1.5.

Листинг 1.1.5. Пример использования директивы `INVOKE`

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;--------------------------------------
;прототип внешней процедуры
PROC1 PROTO
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
INVOKE PROC1
```

```
RET   ;выход
_TEXT ENDS
END START
```

Трансляция программы из листинга 1.1.5:

```
ml /c /coff prog1.asm
link /SUBSYSTEM:WINDOWS prog1.OBJ prog2.OBJ
```

Как видите, внешняя процедура объявляется теперь при помощи директивы `PROTO`. Данная директива позволяет при необходимости указывать и наличие параметров. Например, строка

```
PROC1 PROTO :DWORD, :WORD
```

будет означать, что процедура требует два параметра длиной в четыре и два байта (всего 6, т. е. `@6`).

Как уже говорилось, я буду редко использовать оператор `INVOKE`. Теперь я назову первую причину такого пренебрежения к данной возможности. Дело в том, что я сторонник чистоты языка ассемблера, и любое использование макросредств вызывает у меня чувство несовершенства. На мой взгляд, и начинающим программистам не стоит увлекаться макросредствами, иначе они не почувствуют всю красоту этого языка. О второй причине вы узнаете позже.

На нашей схеме (см. рис. 1.1.1) указано, что существует возможность подсоединения не только объектных модулей, но и библиотек. Собственно, если объектных модулей несколько, то это по понятным причинам вызовет неудобства. Поэтому объектные модули объединяются в библиотеки. Для подсоединения библиотеки в MASM удобнее всего использовать директиву `INCLUDELIB`, которая сохраняется в объектном коде и используется программой LINK.EXE.

Но как создать библиотеку из объектных модулей? Для этого у программы LINK.EXE имеется специальная опция `/LIB`, используемая для управления статическими библиотеками. Предположим, мы хотим создать библиотеку LIB1.LIB, состоящую из одного модуля — PROG2.OBJ. Выполним для этого следующую команду:

```
LINK /lib /OUT:LIB1.LIB PROG2.OBJ
```

Если необходимо добавить в библиотеку еще один модуль (MODUL.OBJ), то достаточно выполнить команду:[5]

```
LINK /LIB LIB1.LIB MODUL.OBJ
```

Вот еще два полезных примера использования библиотекаря:

---

[5] Вместо косой черты для выделения параметра программы LINK.EXE можно использовать черточку, например, так `LINK -LIB LIB1.LIB MODUL.OBJ`.

- `LINK /LIB /LIST LIB1.LIB` — получить список модулей библиотеки;
- `LINK /LIB /REMOVE:MODUL.OBJ LIB1.LIB` — удалить из библиотеки модуль MODUL.OBJ.

Вернемся теперь к нашему примеру. Вместо объектного модуля мы теперь используем библиотеку LIB1.LIB. Видоизмененный текст программы PROG1.ASM представлен в листинге 1.1.6.

Листинг 1.1.6. Пример использования библиотеки

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;----------------------------------------------------------
;прототип внешней процедуры
EXTERN   PROC1@0:NEAR
;----------------------------------------------------------
INCLUDELIB LIB1.LIB
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
CALL PROC1@0
RET  ;выход
_TEXT ENDS
END START
```

Трансляция программы из листинга 1.1.6:

```
ml /c /coff prog1.asm
link /SUBSYSTEM:WINDOWS prog1.OBJ
```

Данные в объектном модуле

Рассмотрим теперь менее важный (для нас) вопрос об использовании данных (переменных), определенных в другом объектном модуле. Здесь читателю, просмотревшему предыдущий материал, должно быть все понятно, а модули PROG2.ASM и PROG1.ASM, демонстрирующие технику использования внешних[6] переменных, приводятся в листингах 1.1.7 и 1.1.8.

---

[6] Термин "внешняя переменная" используется нами по аналогии с термином "внешняя процедура".

Листинг 1.1.7. Модуль, содержащий переменную ALT, которая используется в другом модуле (PROG1.ASM)

```
.586P
;модуль PROG2.ASM
;плоская модель памяти
.MODEL FLAT, STDCALL
PUBLIC PROC1
PUBLIC ALT
;сегмент данных
_DATA SEGMENT
ALT DWORD 0
_DATA ENDS
_TEXT SEGMENT
PROC1 PROC
MOV EAX,ALT
ADD EAX,10
RET
PROC1 ENDP
_TEXT ENDS
END
```

Листинг 1.1.8. Модуль, использующий переменную ALT, определенную в другом модуле (PROG2.ASM)

```
.586P
;модуль PROG1.ASM
;плоская модель памяти
.MODEL FLAT, STDCALL
;---------------------------------------
;прототип внешней процедуры
EXTERN   PROC1@0:NEAR
;внешняя переменная
EXTERN   ALT:DWORD
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
MOV  ALT,10
CALL PROC1@0
MOV  EAX,ALT
RET   ;выход
_TEXT ENDS
END START
```

Замечу, что в отличие от внешних процедур, внешняя переменная[7] не требует добавки `@N`, поскольку размер переменной известен.

Трансляция модулей из листингов 1.1.7 и 1.1.8:

```
ml /c /coff prog2.asm
ml /c /coff prog1.asm
link /subsystem:windows prog1.obj prog2.obj
```

Упрощенный режим сегментации

Ассемблер MASM32 поддерживает так называемую *упрощенную сегментацию*. Я являюсь приверженцем классической структуры ассемблерной программы, но должен признаться, что упрощенная сегментация довольно удобная штука, особенно при программировании под Windows. Суть такой сегментации в следующем: начало сегмента определяется директивой `.CODE`, а сегмента данных — `.DATA`[8]. Причем обе директивы могут появляться в тексте программы несколько раз. Транслятор затем собирает код и данные вместе, как положено. Основной целью такого подхода, по-видимому, является возможность приблизить в тексте программы данные к тем строкам, где они используются. Такая возможность, как известно, в свое время была реализована в C++ (в классическом языке С это было невозможно). На мой взгляд, она приводит к определенному неудобству при чтении текста программы и дальнейшей модификации кода. Кроме того, не сочтите меня за эстета, но когда я вижу данные, перемешанные в тексте программы с кодом, у меня возникает чувство дискомфорта.

В листинге 1.1.9 представлена программа, демонстрирующая упрощенный режим сегментации.

Листинг 1.1.9. Пример программы, использующей упрощенную сегментацию

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;------------------------------------------------------------
;сегмент данных
.DATA
SUM DWORD 0
;сегмент кода
```

[7] То есть (еще раз подчеркну) переменная, определенная в другом модуле.

[8] Разумеется, есть директива и для стека — это `.STACK`, но мы ее почти не будем использовать.

```
.CODE
START:
;сегмент данных
.DATA
A    DWORD 100
;сегмент кода
.CODE
MOV EAX,A
;сегмент данных
.DATA
B    DWORD 200
;сегмент кода
.CODE
ADD EAX,B
MOV SUM,EAX
RET   ;выход
END START
```

Трансляция программы из листинга 1.1.9:

```
ml /c /coff prog.asm
link /subsystem:windows prog.obj
```

> ***Замечание***
>
> Заметим, что макрокоманды `.DATA` и `.CODE` могут использоваться внутри кодового сегмента, определенного традиционным способом. Это удобно для создания разных полезных макроопределений (о макроопределениях подробнее *см. в главе 2.7*).

О пакете MASM32

В данной книге я делаю упор на использование программ пакета MASM32. Последние версии данного пакета можно свободно скачать с сайта **http://www.movsd.com**. Это сайт Стива Хатчессона, создателя пакета MASM32. Данный пакет специально предназначен для создания исполняемого кода для операционной системы Windows и базируется на созданных и поддерживаемых фирмой Microsoft продуктах, таких как транслятор языка ассемблера ML.EXE или редактор связи LINK.EXE.

Пакет MASM32 является свободно распространяемым продуктом, и вы, уважаемые читатели, можете законно использовать его для создания своих программ. Особенностью пакета MASM32 является то, что он ориентирован на создание программ, состоящих из макроопределений и библиотечных про-

цедур. В таком виде программа будет весьма напоминать программу на языке высокого уровня. Такая программа неизбежно также будет содержать и недостатки программ на языках высокого уровня — наличие избыточного кода. Это объясняется очень просто: при создании библиотечных процедур разработчик неизбежно должен добавлять избыточный код, для проведения дополнительных проверок, чтобы создаваемый модуль можно было применять для широкого спектра задач. Избегая использования библиотечных процедур и макросов, можно получить наиболее компактный и производительный код. И хотя я добавил материал, позволяющий читателю ближе познакомиться с возможностями пакета MASM, я по-прежнему остаюсь приверженцем первоначальной концепции книги: учиться писать программы на детальном ассемблере.

Обзор пакета MASM32

Сам пакет распространяется в виде одного исполняемого модуля, в котором в сжатом виде содержатся все компоненты пакета. При инсталляции вы можете выбрать один из локальных дисков, где будет располагаться пакет. В корневом каталоге будет создана папка MASM32, куда и будут помещены все компоненты пакета.

Весь пакет MASM32 состоит из следующих частей (блоков).

- **Инструментальный блок.** Основной инструментарий вы можете найти в подкаталоге \bin. Здесь располагаются программы, которые мы и в дальнейшем будем использовать для трансляции своих программ. Это в первую очередь ассемблер ML.EXE, компоновщик LINK.EXE, а также транслятор ресурсов RC.EXE. О других программах этого каталога мы поговорим в свое время. Из других инструментальных программ следует выделить программу QEDITOR.EXE. Это редактор, который можно с успехом использовать при написании ассемблерных программ. Он обладает рядом полезных свойств, о которых я расскажу далее. Хочется заметить, что для многих утилит пакета вы найдете здесь ассемблерные тексты, с помощью которых сможете менять функциональность программ.
- **Информационный блок.** Следует иметь в виду, что пакет MASM32 не содержит сколько-нибудь полного руководства, которое можно было бы порекомендовать начинающим программистам. Информация, которую вы здесь найдете, предназначена для программистов с некоторым опытом программирования, в том числе и на языке ассемблера. Прежде всего, обратите внимание на подкаталог \help, где располагаются основные файлы этого блока. Они имеют устаревший в настоящее время для файлов помо-

щи формат Winhelp[9] и содержат справочную информацию. Кроме того, некоторые интересные тексты вы найдете в подкаталоге \html. Все они написаны в формате HTML. Отдельные файлы, содержащие много полезной информации, вы сможете найти в самых разных каталогах папки MASM32. Следует также обратить внимание на подкаталоги \icztutes и \tutorial, где можно обнаружить отдельные руководства по многим вопросам программирования на MASM32.

- **Блок примеров.** Пакет содержит огромное количество примеров по самым разным вопросам программирования на MASM под Windows. Они располагаются в основном в подкаталогах: \examples, \com, \oop. Многие каталоги с примерами содержат пакетный файл, с помощью которого вы легко сможете перетранслировать исправленные вами же тексты. "На примерах учатся" — эта педагогическая истина работает уже тысячи лет и, по-видимому, является абсолютной.
- **Блок описаний API-функций.** Блок представляет собой большое количество файлов с расширением inc, расположенных в подкаталоге \include. В файлах содержатся прототипы большинства функций API операционной системы Windows, а также различных полезных констант. Я в своей книге не подключаю к программам эти файлы, а помещаю непосредственно в тексте программ определения функций API и значения полезных констант. Мне кажется, это педагогически оправдано.
- **Блок библиотек.** Располагаются библиотеки в каталоге \lib и имеют расширение lib. Большинство этих библиотек не содержат в себе кода, а являются лишь шлюзами для вызова функций API, код которых располагается в динамических библиотеках, а те, в свою очередь, размещены в подкаталоге \SYSTEM32 каталога Windows. Исключение им составляет, в частности, библиотека MASM32.LIB, которая содержит большое количество процедур, написанных авторами пакета, которые могут значительно облегчить программирование на ассемблере. Тексты этих процедур, с которыми весьма полезно познакомиться, находятся в подкаталоге \M32LIB. Описание библиотечных процедур MASM32.LIB можно найти в подкаталоге \help — файл MASMLIB.HELP. Обращаю также внимание читателей на подкаталог \fpulib, где можно найти тексты библиотеки fpu.lib, в которой располагаются процедуры для манипулирования числами с плавающей точкой.

---

[9] Для использования данного формата помощи в системе должна присутствовать системная утилита winhelp32.exe, которая уже не поставляется с операционной системой Windows Vista. Впрочем, вы можете взять эту утилиту из предыдущих версий Windows.

- **Библиотека макросов.** Библиотека макросов находится в подкаталоге \macros. Тексты макросов располагаются в файле macros.asm. При необходимости использовать макрос этот файл следует подключить к вашей программе при помощи директивы `INCLUDE`.

Трансляторы

С программами ML.EXE и LINK.EXE мы уже познакомились. В *главе 1.4* мы подробнее остановимся на возможностях этих программ. Программы располагаются в подкаталоге \bin. Здесь же в подкаталоге располагается программа RC.EXE — транслятор ресурсов. Эта программа будет нам необходима, если в своем проекте мы будем использовать ресурсы. Я обращаю внимание, что для транслятора ресурсов имеется файл помощи — RC.HLP, где описываются не только возможности программы RC.EXE, но и довольно подробно говорится о структуре самого файла ресурсов. К слову сказать, в каталоге \bin имеется программа IMAGEDIT.EXE, с помощью которой можно создавать графические компоненты ресурсов. Файл же ресурсов, который затем должен быть откомпилирован программой RC.EXE, можно создать в обычном текстовом редакторе *(см. главу 2.4)* или при помощи специализированного редактора ресурсов, который можно найти, например, в пакете Visual Studio .NET.

В пакет MASM32 были включены также еще три программы, которые также можно использовать при трансляции: POLINK.EXE — редактор связей, POLIB.EXE — программа-библиотекарь, PORC.EXE — транслятор ресурсов. Программы располагаются все в том же подкаталоге \bin, их предоставил создателям пакета MASM автор Пэл Ориниус (префикс PO). Вы можете использовать эти программы вместо традиционных программ LINK.EXE и RC.EXE от Microsoft.

Редактор QEDITOR

Редактор QEDITOR.EXE, который поставляется вместе с пакетом MASM32, располагается непосредственно в корневом каталоге пакета. Сам редактор и все сопутствующие ему утилиты написаны на ассемблере. Анализ их размера и возможностей действительно впечатляет. Например, сам редактор имеет длину всего 37 Кбайт.

Редактор вполне годится для работы с небольшими одномодульными приложениями. Для работы с несколькими модулями он не очень удобен. Работа редактора основана на взаимодействии с различными утилитами посредством пакетных файлов. Например, трансляцию программ осуществляет пакетный файл ASSMBL.BAT, который использует ассемблер ML.EXE, а результат

ассемблирования направляется в текстовый файл ASMBL.TXT. Далее для просмотра этого файла используется простая утилита THEGUN.EXE (кстати, размером всего 6 Кбайт). Аналогично осуществляется редактирование связей. Для дизассемблирования исполняемого модуля служит утилита DUMPPE.EXE, результат работы этой утилиты помещается в текстовый файл DISASM.TXT. Аналогично осуществляются и другие операции. Вы легко сможете настроить эти операции, отредактировав соответствующий пакетный файл с модификацией (при необходимости) используемых утилит (заменив, например, ML.EXE на TASM32.EXE и т. п.).

Редактор обладает одной важной и полезной особенностью: при помощи пункта **Edit | Edit Menus** вы можете удалять или добавлять пункты меню (рис. 1.1.2). Синтаксис описания меню довольно прост, и вы легко сможете в нем самостоятельно разобраться и добавлять свои пункты меню, привязывая их к тем или иным утилитам или файлам иного формата.

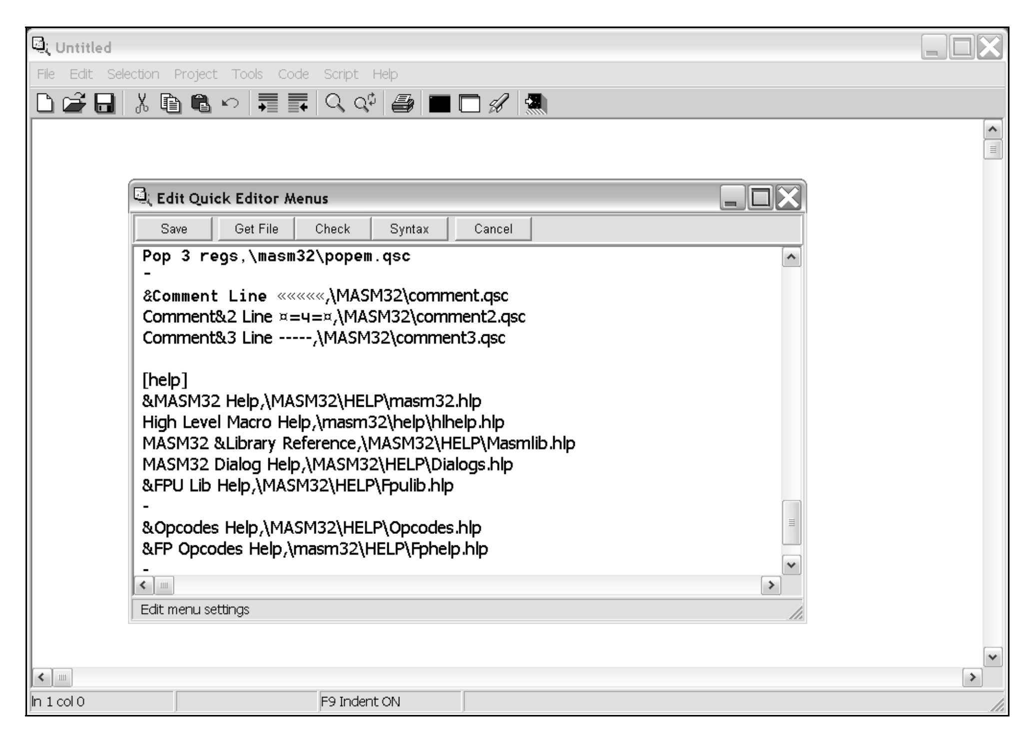


Рис. 1.1.2. Окно редактора QEDITOR.EXE вместе с окном редактирования меню

Обратите также внимание на пункты меню **Code** и **Script**. Первый пункт меню содержит, в частности, команды генерации шаблонов простых приложений.

Вы можете сгенерировать такой шаблон, а затем по своему усмотрению наращивать его функциональность. В пункте **Script** находятся команды генерации некоторых полезных программных блоков, например блок SWITCH. Сценарии генерации содержатся здесь же в каталоге и имеют простой текстовый формат и расширение qsc.

Дизассемблеры

Дизассемблеры переводят исполняемый модуль в ассемблерный код. Примером простейшего дизассемблера является программа DUMPPE.EXE, работающая в строковом режиме. Пример работы программы DUMPPE.EXE представлен в листинге 1.1.10 (имя дизассемблируемой программы следует указать в командной строке). Здесь дизассемблирована программа, приведенная в листинге 1.1.5. Ну, как, узнали? Смысл обозначений будет ясен из *части IV* (см. также [27]). Замечу только, что большая часть листинга состоит из описания заголовка исполняемого модуля. Дизассемблированный же текст начинается после ключевого слова `disassembly`. В *части IV* мы будем подробно рассматривать и сами дизассемблеры, и методику их использования.

Листинг 1.1.10. Пример дизассемблирования программы с помощью DUMPPE.EXE

```
1-5.exe                             (hex)           (dec)

.EXE size (bytes)                     490            1168
Minimum load size (bytes)             450            1104
Overlay number                          0               0
Initial CS:IP                   0000:0000
Initial SS:SP                   0000:00B8             184
Minimum allocation (para)               0               0
Maximum allocation (para)            FFFF           65535
Header size (para)                      4               4
Relocation table offset                40              64
Relocation entries                      0               0

Portable Executable starts at                   a8
Signature                                 00004550 (PE)
Machine                                       014C (Intel 386)
Sections                                      0001
Time Date Stamp                           4571CA64 Sat Dec  2 23:48:04 2006
Symbol Table                              00000000
Number of Symbols                         00000000
Optional header size                          00E0
Characteristics                               010F
```

```
       Relocation information stripped
       Executable Image
       Line numbers stripped
       Local symbols stripped
       32 bit word machine
Magic                                       010B
Linker Version                              5.12
Size of Code                            00000200
Size of Initialized Data                00000000
Size of Uninitialized Data              00000000
Address of Entry Point                  00001000
Base of Code                            00001000
Base of Data                            00002000
Image Base                              00400000
Section Alignment                       00001000
File Alignment                          00000200
Operating System Version                    4.00
Image Version                               0.00
Subsystem Version                           4.00
reserved                                00000000
Image Size                              00002000
Header Size                             00000200
Checksum                                00000000
Subsystem                                   0002 (Windows)
DLL Characteristics                         0000
Size Of Stack Reserve                   00100000
Size Of Stack Commit                    00001000
Size Of Heap Reserve                    00100000
Size Of Heap Commit                     00001000
Loader Flags                            00000000
Number of Directories                   00000010

Directory Name                          VirtAddr  VirtSize
--------------------------------------  --------  --------
Export                                  00000000  00000000
Import                                  00000000  00000000
Resource                                00000000  00000000
Exception                               00000000  00000000
Security                                00000000  00000000
Base Relocation                         00000000  00000000
Debug                                   00000000  00000000
Decription/Architecture                 00000000  00000000
Machine Value (MIPS GP)                 00000000  00000000
Thread Storage                          00000000  00000000
Load Configuration                      00000000  00000000
Bound Import                            00000000  00000000
```

```
Import Address Table                  00000000  00000000
Delay Import                          00000000  00000000
COM Runtime Descriptor                00000000  00000000
(reserved)                            00000000  00000000

Section Table
-------------
01  .text     Virtual Address       00001000
              Virtual Size          00000016
              Raw Data Offset       00000200
              Raw Data Size         00000200
              Relocation Offset     00000000
              Relocation Count      0000
              Line Number Offset    00000000
              Line Number Count     0000
              Characteristics       60000020
                     Code
                     Executable
                     Readable

Disassembly

00401000                     start:
00401000 E80B000000              call    fn_00401010
00401005 C3                      ret
00401006 CC                      int     3
00401007 CC                      int     3
00401008 CC                      int     3
00401009 CC                      int     3
0040100A CC                      int     3
0040100B CC                      int     3
0040100C CC                      int     3
0040100D CC                      int     3
0040100E CC                      int     3
0040100F CC                      int     3
00401010                     fn_00401010:
00401010 B8E8030000              mov     eax,3E8h
00401015 C3                      ret
```

Если вас интересует заголовок исполняемого модуля, то вместо утилиты DUMPE.EXE можно использовать ключ `/dump` для утилиты LINK.EXE. Например, выполнив для программы 1-5.exe командную строку `LINK /dump /all 1-5.exe`, получим листинг 1.1.11. В нем программная часть модуля представлена в виде таблицы, состоящей из шестнадцатеричных кодов, т. е. дампа.

Листинг 1.1.11. Результат работы программы LINK.EXE с опцией `/dump`

```
Microsoft (R) COFF Binary File Dumper Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file 1-5.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
             14C machine (i386)
               1 number of sections
        4571CA64 time date stamp Sat Dec 02 23:48:04 2006
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
             10F characteristics
                   Relocations stripped
                   Executable
                   Line numbers stripped
                   Symbols stripped
                   32 bit word machine

OPTIONAL HEADER VALUES
             10B magic #
            5.12 linker version
             200 size of code
               0 size of initialized data
               0 size of uninitialized data
            1000 RVA of entry point
            1000 base of code
            2000 base of data
          400000 image base
            1000 section alignment
             200 file alignment
            4.00 operating system version
            0.00 image version
            4.00 subsystem version
               0 Win32 version
            2000 size of image
             200 size of headers
               0 checksum
               2 subsystem (Windows GUI)
               0 DLL characteristics
```

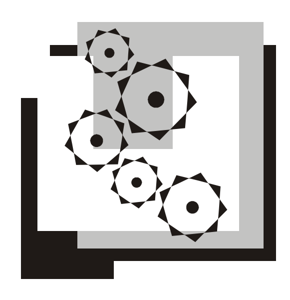
```
          100000 size of stack reserve
            1000 size of stack commit
          100000 size of heap reserve
            1000 size of heap commit
               0 loader flags
              10 number of directories
               0 [       0] RVA [size] of Export Directory
               0 [       0] RVA [size] of Import Directory
               0 [       0] RVA [size] of Resource Directory
               0 [       0] RVA [size] of Exception Directory
               0 [       0] RVA [size] of Certificates Directory
               0 [       0] RVA [size] of Base Relocation Directory
               0 [       0] RVA [size] of Debug Directory
               0 [       0] RVA [size] of Architecture Directory
               0 [       0] RVA [size] of Special Directory
               0 [       0] RVA [size] of Thread Storage Directory
               0 [       0] RVA [size] of Load Configuration Directory
               0 [       0] RVA [size] of Bound Import Directory
               0 [       0] RVA [size] of Import Address Table Directory
               0 [       0] RVA [size] of Delay Import Directory
               0 [       0] RVA [size] of Reserved Directory
               0 [       0] RVA [size] of Reserved Directory

SECTION HEADER #1
   .text name
      16 virtual size
    1000 virtual address
     200 size of raw data
     200 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         Execute Read

RAW DATA #1
00401000: E8 0B 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC  ш....├╠╠╠╠╠╠╠╠╠╠
00401010: B8 E8 03 00 00 C3                                ╕ш...├

  Summary

        1000 .text
```



Глава 1.2

Основы программирования в операционной системе Windows

В данной главе я намерен рассмотреть два вопроса, которые крайне важны для того, чтобы начать программировать на ассемблере в среде Windows — это вызов системных функций (API-функций) и возможные структуры программ[1] для Windows. Я полагаю, что можно выделить шесть типов структур (каркасов) программ, которые условно можно назвать:[2]

- классическая структура — имеет одно главное окно;
- диалоговая структура — главным окном является диалоговое *модальное* окно;
- консольный тип — главным окном является консольное окно (создаваемое или наследуемое);
- безоконная[3] структура — это Windows-приложение, не имеющее окон;
- сервисы — специальные программы, играющие особую роль в операционной системе, которые могут автоматически загружаться при запуске системы;
- драйверы — специальные программы управления внешними устройствами и имеющими привилегированный доступ к ресурсам операционной системы.

В данной главе подробно описывается первая, классическая структура программ.

---

[1] Не путать со структурой загружаемых модулей.
[2] Классификация автора.
[3] Как потом станет ясно, консольное приложение вполне может иметь диалоговое окно, а оконная программа — консольное окно.

Итак, начнем с нескольких общих положений о программировании в Windows. Те, кто уже имеет опыт программирования в среде Windows, могут на этом не останавливаться.

- Программирование в Windows основывается на использовании функций API (Application Program Interface, интерфейс программного приложения). Их количество в операционной системе Windows достигает почти четырех тысяч. Ваша программа в значительной степени будет состоять из таких вызовов. Все взаимодействие с внешними устройствами и ресурсами операционной системы будет происходить посредством таких функций.
- Список функций API и их описание лучше всего брать из файла WIN32.HLP, который поставляется, например, с пакетом Borland C++ или Delphi. Подробнейшее описание по функциям API и вообще по программированию в Windows содержится в документации к Visual Studio .NET — это информация из первых рук!
- Главным элементом программы в среде Windows является *окно*. Для каждого окна определяется своя процедура[4] обработки сообщений *(см. далее)*. Программирование в Windows в значительной степени заключается в программировании оконных процедур.
- Окно может содержать элементы управления: кнопки, списки, окна редактирования и др. Эти элементы, по сути, также являются окнами, но обладающими особыми свойствами, которые автоматически поддерживает операционная система. События, происходящие с этими элементами (и самим окном), способствуют приходу сообщений в процедуру окна.
- Операционная система Windows использует *линейную адресацию памяти*. Другими словами, всю память конкретной задачи (процесса) можно рассматривать как один большой линейный блок. Для программиста на языке ассемблера это означает, что адрес любой ячейки памяти будет определяться содержимым одного 32-битного регистра, например `EBX`.
- Следствием предыдущего пункта является то, что мы фактически не ограничены в объеме данных, кода или стека (объеме локальных переменных). Сегменты в тексте программы играют роль определения секций исполняемого кода, которые обладают определенными свойствами: запрет на запись, общий доступ и т. д.
- Операционная система Windows является многозадачной средой. Каждая задача имеет свое адресное пространство и свою очередь сообщений.

---

[4] Исходя из терминологии, принятой когда-то в операционной системе MS-DOS, такую процедуру следует назвать "процедурой прерывания". Для Windows же принята другая терминология. Подобные процедуры, вызываемые самой системой, называются процедурами обратного вызова (callback).

Более того, даже в рамках одной программы может быть осуществлена многозадачность — любая процедура может быть запущена как самостоятельная задача (правильнее назвать это потоком).

Итак, рассмотрев общие положения, перейдем к практическим примерам.

Вызов функций API

Начнем с того, что рассмотрим, как можно вызвать функции API. Обратимся к файлу помощи и выберем любую функцию API, например, `MessageBox`. Вот описание функции в нотации языка С:

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

Данная функция выводит на экран окно с сообщением и кнопкой (или кнопками) выхода. Смысл параметров функции: *`hWnd`* — дескриптор окна, в котором будет появляться окно-сообщение, *`lpText`* — текст, который будет появляться в окне, *`lpCaption`* — текст в заголовке окна, *`uType`* — тип окна, в частности можно определить количество кнопок выхода. Теперь о типах указанных параметров. Все они в действительности 32-битные целые числа: `HWND` — 32-битное целое, `LPCTSTR` — 32-битный указатель на строку, `UINT` — 32-битное целое. К имени функций нам придется добавлять суффикс "A". Это означает, что строковые параметры в данной функции должны иметь кодировку в стандарте ANSI. Добавление в конце функции суффикса "W" будет означать, что все строковые параметры будут иметь кодировку Unicode *(подробнее см. главу 1.5)*. Кроме того, при использовании MASM необходимо также в конце имени добавить `@16` (4 параметра по 4 байта). Таким образом, вызов указанной функции будет выглядеть так: `CALL MessageBoxA@16`. А как же быть с параметрами? Их следует предварительно аккуратно поместить в стек командой `PUSH`. Запомните "золотое" правило: **СЛЕВА НАПРАВО — СНИЗУ ВВЕРХ**. Итак, пусть дескриптор окна расположен по адресу `HW`, строки — по адресам `STR1` и `STR2`, а тип окна-сообщения — это константа. Самый простой тип сообщения имеет значение 0 и называется `MB_OK`. Это сообщение предполагает окно с одной кнопкой. Имеем следующую схему вызова:

```
MB_OK   equ 0
.
.
STR1    DB "Неверный ввод! ",0
STR2    DB "Сообщение об ошибке. ",0
HW      DWORD ?
.
.
```

```
PUSH    MB_OK
PUSH    OFFSET STR1
PUSH    OFFSET STR2
PUSH    HW
CALL    MessageBoxA@16
```

Как видите, все весьма просто и ничуть не сложнее, как если бы вы вызывали эту функцию на C или Delphi. Результат выполнения любой функции — это, как правило, целое число, которое возвращается в регистре EAX.

Особо следует остановиться на строках. Вообще, когда говорят о строках, то имеют в виду обычно адрес (или указатель) строки. В большинстве случаев предполагается, что строка заканчивается символом с кодом 0. Однако довольно часто один из параметров функции API задает длину строки (в документации такая строка часто называется буфером), определяемой другим параметром. Это надо иметь в виду, потому что может оказаться, что возвращаемая строка не имеет нуля на конце, который необходим для использования ее в другой функции. Неучет этого момента может привести к появлению в программе фантомных ошибок, т. е. ошибок, которые подобно призраку могут появляться и исчезать.

> ***Замечание***
>
> В документации Microsoft утверждает, что при возвращении из функции API должны сохраниться регистры EBX, EBP, ESP, ESI, EDI. Значение функции, как обычно, возвращается в регистр EAX. Сохранность содержимого других регистров не гарантируется.

Аналогичным образом в ассемблере легко воспроизвести те или иные C-структуры. Рассмотрим, например, структуру, определяющую системное сообщение:

```
typedef struct tagMSG {  // msg
HWND hwnd;
UINT message;
WPARAM wParam;
LPARAM lParam;
DWORD time;
POINT pt;
} MSG;
```

Это сообщение будет далее подробно прокомментировано в одном из примеров. На MASM эта структура будет иметь вид:

```
MSGSTRUCT STRUC
       MSHWND         DD ?
       MSMESSAGE      DD ?
       MSWPARAM       DD ?
       MSLPARAM       DD ?
```

```
        MSTIME          DD ?
        MSPT            DD ?
MSGSTRUCT ENDS
```

Как видите, на ассемблере все даже гораздо проще. Вообще, трудно не отклониться от темы и еще раз не удивиться фирме Microsoft, все так усложнившей в отношении типов переменных, используемых при программировании в Windows.

> ***Замечание***
>
> Особо отмечу, что при вызове функции API может возникнуть ошибка, связанная либо с неправильными входными параметрами, либо невозможностью получения искомых результатов из-за определенного состояния системы. В этом случае индикатором будет содержимое, возвращаемое в регистре `EAX`. К сожалению, для разных функций это может быть разное значение. Это может быть 0 (чаще всего), –1 или какое-либо другое ненулевое значение. Напомню в этой связи, что 0 во многих языках программирования считается синонимом значения `FALSE`, а значению `TRUE` ставится в соответствие 1. В каждом конкретном случае следует свериться с документацией. С помощью функции `GetLastError` можно получить код последней происшедшей ошибки, т. е. расшифровку того, почему функция API выполнилась не должным образом. Использование функции `GetLastError` вы найдете в дальнейших наших примерах. Для расшифровки кода ошибки удобно воспользоваться программой errlook.exe из пакета Microsoft Visual Studio .NET. Программа принимает код ошибки и возвращает текстовый комментарий к ошибке.

Структура программы

Теперь обратимся к структуре всей программы. Как я уже говорил, в данной главе мы будем рассматривать классическую структуру программы под Windows. В такой программе имеется главное окно, а следовательно, и процедура главного окна. В целом, в коде программы можно выделить следующие секции:

- регистрация класса окон;
- создание главного окна;
- цикл обработки очереди сообщений;
- процедура главного окна.

Конечно, в программе могут быть и другие разделы, но данные разделы образуют основной каркас программы. Разберем эти разделы по порядку.

Регистрация класса окон

Регистрация класса окон осуществляется с помощью функции `RegisterClassA`, единственным параметром которой является указатель на структуру `WNDCLASS`, содержащую информацию об окне (см. листинг 1.2.1).

Создание окна

На основе зарегистрированного класса с помощью функции CreateWindowExA (или CreateWindowA) можно создать экземпляр окна. Как можно заметить, это весьма напоминает объектную модель программирования. По этой причине объектная модель легко строится для оконной операционной системы Windows.

Цикл обработки очереди сообщений

Вот как обычно выглядит этот цикл на языке С:

```
 while (GetMessage (&msg,NULL,0,0))
 {
   // Разрешить использование клавиатуры
   // путем трансляции сообщений о виртуальных клавишах
   // в сообщения об алфавитно-цифровых клавишах
   TranslateMessage(&msg);
   // Вернуть управление Windows и передать сообщение дальше
   // процедуре окна
   DispatchMessage(&msg);
}
```

Функция GetMessage() "отлавливает" очередное сообщение из ряда сообщений данного приложения и помещает его в структуру MSG. Если сообщений в очереди нет, то функция ждет появление сообщения. Вместо функции GetMessage часто используют функцию PeekMessage с тем же набором параметров. Отличие функции GetMessage от PeekMessage заключается в том, что последняя функция не ожидает сообщения, если его нет в очереди. Функцию PeekMessage часто используют, чтобы несколько оптимизировать работу программы.

Что касается функции TranslateMessage, то ее компетенция касается сообщений WM_KEYDOWN и WM_KEYUP, которые транслируются в WM_CHAR и WM_DEDCHAR, а также WM_SYSKEYDOWN и WM_SYSKEYUP, преобразующиеся в WM_SYSCHAR и WM_SYSDEADCHAR. Смысл трансляции заключается не в замене, а в отправке дополнительных сообщений. Так, например, при нажатии и отпускании алфавитно-цифровой клавиши в окно сначала придет сообщение WM_KEYDOWN, затем WM_KEYUP, а затем уже WM_CHAR.

Как можно видеть, выход из цикла ожиданий имеет место только в том случае, если функция GetMessage возвращает 0. Это происходит только при получении сообщения о выходе (сообщение WM_QUIT, см. далее). Таким образом, цикл ожидания играет двоякую роль: определенным образом преобразуются сообщения, предназначенные для какого-либо окна, и ожидается сообщение о выходе из программы.

Процедура главного окна

Вот прототип функции[5] окна на языке С:

```
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
```

Оставив в стороне тип возвращаемого функцией значения[6], обратите внимание на передаваемые параметры. Вот смысл этих параметров: *hwnd* — идентификатор окна, *message* — идентификатор сообщения, *wParam* и *lParam* — параметры, уточняющие смысл сообщения (для каждого сообщения они могут играть разные роли или не играть никаких). Все четыре параметра, как вы, наверное, уже догадались, имеют тип DWORD.

А теперь рассмотрим каркас этой функции на языке ассемблера (листинг 1.2.1).

Листинг 1.2.1. Каркас оконной процедуры

```
WNDPROC PROC
      PUSH EBP
      MOV EBP,ESP ; теперь EBP указывает на вершину стека
      PUSH EBX
      PUSH ESI
      PUSH EDI
      PUSH DWORD PTR [EBP+14H]   ; LPARAM (lParam)
      PUSH DWORD PTR [EBP+10H]   ; WPARAM (wParam)
      PUSH DWORD PTR [EBP+0CH]   ; MES (message)
      PUSH DWORD PTR [EBP+08H]   ; HWND (hwnd)
      CALL DefWindowProcA@16
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      RET 16
WNDPROC ENDP
```

Прокомментируем фрагмент из листинга 1.2.1.

RET 16 — возврат из процедуры с освобождением стека от четырех параметров ($16 = 4 \times 4$).

Доступ к параметрам осуществляется через регистр EBP:

```
DWORD PTR [EBP+14H]   ; LPARAM (lParam)
DWORD PTR [EBP+10H]   ; WPARAM (wParam)
```

[5] Напоминаю, что в данной книге термины "процедура" и "функция" в части программ на языке ассемблера являются синонимами.

[6] Нам это никогда не понадобится.

```
DWORD PTR [EBP+0CH]   ; MES (message) — код сообщения
DWORD PTR [EBP+08H]   ; HWND (hwnd) — дескриптор окна
```

Функция `DefWindowProc` вызывается для тех сообщений, которые не обрабатываются в функции окна. В данном примере, как вы понимаете, не обрабатываются все приходящие в функцию окна сообщения. Как видно, мы гарантировали сохранность четырех регистров: `EBX`, `EBP`, `ESI`, `EDI`. Этого требует документация, хотя я обычно стараюсь, чтобы не изменялись все регистры.

> ***Замечание***
>
> В случае если сообщение обрабатывается процедурой окна, стандартным возвращаемым значением является значение 0 (`FALSE`). Однако бывают и специальные случаи, когда требуется возвращать 1 или –1. Поэтому следует внимательно изучить документацию по каждому обрабатываемому процедурой окна сообщению.

Примеры простых программ для Windows

Приступим теперь к разбору конкретного примера. В листинге 1.2.2 представлена простая программа. Изучите ее внимательно — она является фундаментом, на котором мы будем строить дальнейшее рассмотрение. Окно, появляющееся при запуске этой программы, изображено на рис. 1.2.1.

Прежде всего, обратите внимание на директивы `INCLUDELIB`. В пакете MASM32 довольно много разных библиотек. Для данного примера нам понадобились две: user32.lib и kernel32.lib.

Сначала мы определяем константы и внешние библиотечные процедуры. В действительности все эти определения можно найти в include-файлах, прилагаемых к пакету MASM32. Мы же не будем использовать стандартные include-файлы по двум причинам: во-первых, так удобнее понять технологию программирования, во-вторых, легче перейти от MASM к другим ассемблерам.

Необходимо четко понимать способ функционирования процедуры окна, т. к. именно это определяет суть программирования под Windows. Задача данной процедуры — правильная реакция на все приходящие сообщения. Давайте детально разберемся в работе нашего приложения. Сначала обратим внимание на то, что необработанные сообщения должны возвращаться в систему при помощи функции `DefWindowProcA`. Мы отслеживаем пять сообщений: `WM_CREATE`, `WM_CLOSE`, `WM_DESTROY`, `WM_LBUTTONDOWN`, `WM_RBUTTONDOWN`. Сообщения `WM_CREATE` и `WM_DESTROY` в терминах объектного программирования играют роли конструктора и деструктора: они приходят в функцию окна при создании окна и при уничтожении окна. Если щелкнуть по кнопке-крестику в правом углу окна, то в функцию окна придет сообщение `WM_CLOSE`, а после закрытия

окна — сообщение WM_DESTROY. Это очень важный момент. Сообщение WM_CLOSE, таким образом, сообщает приложению, что пользователь намерен закрыть окно. Но возникает законный вопрос: всякое ли закрытие окна должно привести к закрытию самого приложения? Нет, только при условии, что речь идет о главном окне приложения. У нас как раз тот случай. И именно поэтому, несмотря на то, что сообщение WM_CLOSE нами обработано, мы все равно далее запускаем функцию DefWindowProc. Именно она обеспечивает закрытие окна, а затем приход на функцию окна сообщения WM_DESTROY. Чтобы проверить данное утверждение, достаточно проделать маленький эксперимент. Замените фрагмент, обработки сообщения WM_CLOSE на следующий:

```
WMCLOSE:
        PUSH    0                       ; MB_OK
        PUSH    OFFSET CAP
        PUSH    OFFSET MES3
        PUSH    DWORD PTR [EBP+08H]     ; дескриптор окна
        CALL    MessageBoxA@16
        MOV     EAX,0
        JMP     FINISH
```

Как видите, мы просто обошли вызов функции DefWindowProc. В результате у нас перестанет закрываться окно и, естественно, само приложение: крестик в правом верхнем углу окна перестал работать.

После прихода в функцию окна сообщения WM_DESTROY (а оно приходит уже после того, как окно закрыто) будет выполнена функция PostQuitMessage, и приложению будет послано сообщение WM_QUIT, которое вызовет выход из цикла ожидания и выполнение функции ExitProcess, что в свою очередь приведет к удалению приложения из памяти. Замечу также, что переход на метку WMDESTROY при щелчке правой кнопкой мыши приводит к закрытию приложения. Разумеется, при выходе из приложения операционная система удаляет из памяти все, что было как-то связано с данным приложением, в том числе и главное окно.

Обращаю ваше внимание на метку _ERR — переход на нее происходит при возникновении ошибки, и здесь можно поместить соответствующее сообщение.

Наконец, еще один важный момент. Если ваша оконная процедура обрабатывает какое-либо сообщение, то она должна возвратить 0. Другими словами, нулевое значение следует поместить в регистр EAX. В частности, если после обработки сообщения WM_CREATE возвратить значение –1, то окно не будет создано, а создающая его функция (CreateWindowEx) возвратит 0.

Замечание

Есть еще одно сообщение, которое приходит перед сообщением WM_CREATE — это сообщение WM_NCCREATE. После его обработки следует возвращать 1. Если возвратить 0, то окно не будет создано, а функция CreateWindowEx возвратит 0.

Листинг 1.2.2. Простой пример программы для Windows (MASM32)

```
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
;сообщение приходит при закрытии окна
WM_CLOSE            equ 10h
;сообщение приходит при закрытии окна
WM_DESTROY          equ 2
;сообщение приходит при создании окна
WM_CREATE           equ 1
;сообщение при щелчке левой кнопкой мыши в области окна
WM_LBUTTONDOWN      equ 201h
;сообщение при щелчке правой кнопкой мыши в области окна
WM_RBUTTONDOWN      equ 204h
;свойства окна
CS_VREDRAW          equ 1h
CS_HREDRAW          equ 2h
CS_GLOBALCLASS      equ 4000h
WS_OVERLAPPEDWINDOW equ 000CF0000H
style equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
;идентификатор стандартной пиктограммы
IDI_APPLICATION     equ 32512
;идентификатор курсора
IDC_CROSS           equ 32515
;режим показа окна - нормальный
SW_SHOWNORMAL       equ 1
;прототипы внешних процедур
EXTERN  MessageBoxA@16:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
```

```
;-----------------------------------------------
;структуры
;структура сообщения
MSGSTRUCT STRUC
       MSHWND        DD ? ;идентификатор окна, получающего сообщение
       MSMESSAGE     DD ? ;идентификатор сообщения
       MSWPARAM      DD ? ;дополнительная информация о сообщении
       MSLPARAM      DD ? ;дополнительная информация о сообщении
       MSTIME        DD ? ;время посылки сообщения
       MSPT          DD ? ;положение курсора, во время посылки сообщения
MSGSTRUCT ENDS
;---------
WNDCLASS STRUC
       CLSSTYLE      DD ? ;стиль окна
       CLWNDPROC     DD ? ;указатель на процедуру окна
       CLSCEXTRA     DD ? ;информация о дополнительных байтах
                          ;для данной структуры
       CLWNDEXTRA    DD ? ;информация о дополнительных байтах для окна
       CLSHINSTANCE  DD ? ;дескриптор приложения
       CLSHICON      DD ? ;идентификатор пиктограммы окна
       CLSHCURSOR    DD ? ;идентификатор курсора окна
       CLBKGROUND    DD ? ;идентификатор кисти окна
       CLMENUNAME    DD ? ;имя-идентификатор меню
       CLNAME        DD ? ;специфицирует имя класса окон
WNDCLASS ENDS
;сегмент данных
_DATA SEGMENT
       NEWHWND  DD 0
       MSG           MSGSTRUCT <?>
       WC            WNDCLASS  <?>
       HINST         DD 0 ;здесь хранится дескриптор приложения
       TITLENAME     DB 'Простой пример 32-битного приложения',0
       CLASSNAME     DB 'CLASS32',0
       CAP           DB 'Сообщение',0
       MES1          DB 'Вы нажали левую кнопку мыши',0
       MES2          DB 'Выход из программы. Пока!',0
       MES3          DB 'Закрытие окна',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
       PUSH          0
       CALL          GetModuleHandleA@4
       MOV           HINST, EAX
REG_CLASS:
```

```
;заполнить структуру окна
; стиль
       MOV    WC.CLSSTYLE,style
;процедура обработки сообщений
       MOV    WC.CLWNDPROC, OFFSET WNDPROC
       MOV    WC.CLSCEXTRA, 0
       MOV    WC.CLWNDEXTRA, 0
       MOV    EAX, HINST
       MOV    WC.CLSHINSTANCE, EAX
;----------пиктограмма окна
       PUSH   IDI_APPLICATION
       PUSH   0
       CALL   LoadIconA@8
       MOV    WC.CLSHICON, EAX
;----------курсор окна
       PUSH   IDC_CROSS
       PUSH   0
       CALL   LoadCursorA@8
       MOV    WC.CLSHCURSOR, EAX
;----------
       MOV    WC.CLBKGROUND, 17 ;цвет окна
       MOV    DWORD PTR WC.CLMENUNAME,0
       MOV    DWORD PTR WC.CLNAME, OFFSET CLASSNAME
       PUSH   OFFSET WC
       CALL   RegisterClassA@4
;создать окно зарегистрированного класса
       PUSH   0
       PUSH   HINST
       PUSH   0
       PUSH   0
       PUSH   400    ; DY - высота окна
       PUSH   400    ; DX - ширина окна
       PUSH   100    ; Y-координата левого верхнего угла
       PUSH   100    ; X-координата левого верхнего угла
       PUSH   WS_OVERLAPPEDWINDOW
       PUSH   OFFSET TITLENAME ;имя окна
       PUSH   OFFSET CLASSNAME ;имя класса
       PUSH   0
       CALL   CreateWindowExA@48
;проверка на ошибку
       CMP   EAX,0
       JZ    _ERR
       MOV   NEWHWND, EAX    ; дескриптор окна
;--------------------------------
       PUSH   SW_SHOWNORMAL
       PUSH   NEWHWND
```

```
       CALL   ShowWindow@8   ; показать созданное окно
;--------------------------------
       PUSH   NEWHWND
       CALL   UpdateWindow@4 ; команда перерисовать видимую
                             ; часть окна, сообщение WM_PAINT
;цикл обработки сообщений
MSG_LOOP:
       PUSH   0
       PUSH   0
       PUSH   0
       PUSH   OFFSET MSG
       CALL   GetMessageA@16
       CMP    EAX, 0
       JE     END_LOOP
       PUSH   OFFSET MSG
       CALL   TranslateMessage@4
       PUSH   OFFSET MSG
       CALL   DispatchMessageA@4
       JMP    MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
       PUSH   MSG.MSWPARAM
       CALL   ExitProcess@4
_ERR:
       JMP    END_LOOP
;-----------------------------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H] LPARAM
; [EBP+10H] WAPARAM
; [EBP+0CH] MES
; [EBP+8] HWND
WNDPROC PROC
       PUSH   EBP
       MOV    EBP, ESP
       PUSH   EBX
       PUSH   ESI
       PUSH   EDI
       CMP    DWORD PTR [EBP+0CH], WM_DESTROY
       JE     WMDESTROY
       CMP    DWORD PTR [EBP+0CH],WM_CLOSE       ;закрытие окна
       JE     WMCLOSE
       CMP    DWORD PTR [EBP+0CH], WM_CREATE
       JE     WMCREATE
       CMP    DWORD PTR [EBP+0CH],WM_LBUTTONDOWN  ;левая кнопка
       JE     LBUTTON
```

```
        CMP     DWORD PTR [EBP+0CH],WM_RBUTTONDOWN  ;правая кнопка
        JE      RBUTTON
        JMP     DEFWNDPROC
;нажатие правой кнопки мыши приводит к закрытию окна
RBUTTON:
        JMP     WMDESTROY
;нажатие левой кнопки мыши
LBUTTON:
;выводим сообщение
        PUSH    0 ;MB_OK
        PUSH    OFFSET CAP
        PUSH    OFFSET MES1
        PUSH    DWORD PTR [EBP+08H]
        CALL    MessageBoxA@16
        MOV     EAX, 0
        JMP     FINISH
WMCREATE:
        MOV     EAX, 0
        JMP     FINISH
WMCLOSE:
        PUSH    0         ;MB_OK
        PUSH    OFFSET CAP
        PUSH    OFFSET MES3
        PUSH    DWORD PTR [EBP+08H] ;дескриптор окна
        CALL    MessageBoxA@16
DEFWNDPROC:
        PUSH    DWORD PTR [EBP+14H]
        PUSH    DWORD PTR [EBP+10H]
        PUSH    DWORD PTR [EBP+0CH]
        PUSH    DWORD PTR [EBP+08H]
        CALL    DefWindowProcA@16
        JMP     FINISH
WMDESTROY:
        PUSH    0         ;MB_OK
        PUSH    OFFSET CAP
        PUSH    OFFSET MES2
        PUSH    DWORD PTR [EBP+08H] ;дескриптор окна
        CALL    MessageBoxA@16
;отправляем сообщение WM_QUIT
        PUSH    0
        CALL    PostQuitMessage@4
        MOV     EAX, 0
FINISH:
        POP     EDI
        POP     ESI
        POP     EBX
```

```
        POP     EBP
        RET     16
WNDPROC ENDP
_TEXT ENDS
END START
```

Трансляция программы:

```
ml /c /coff prog.asm
link /subsystem:windows prog.obj
```

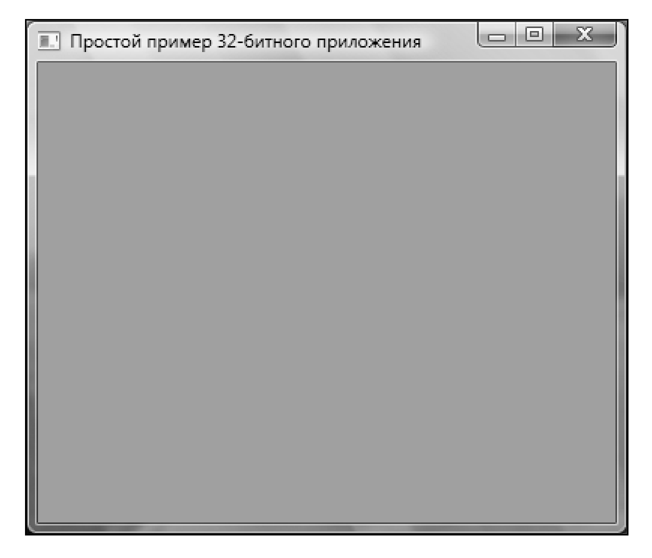


Рис. 1.2.1. Окно программы из листинга 1.2.2

Еще о цикле обработки сообщений

В силу значимости данного элемента оконного приложения я позволю себе еще раз обратиться к циклу обработки сообщений. Вот фрагмент программы из листинга 1.2.2:

```
MSG_LOOP:
        PUSH    0
        PUSH    0
        PUSH    0
        PUSH    OFFSET MSG
        CALL    GetMessageA@16
        CMP     EAX, 0
        JE      END_LOOP
        PUSH    OFFSET MSG
```

```
        CALL    TranslateMessage@4
        PUSH    OFFSET MSG
        CALL    DispatchMessageA@4
        JMP     MSG_LOOP
END_LOOP:
```

Как я уже упоминал, функция `GetMessage` ожидает нового сообщения. При этом, по сути, само приложение бездействует, т. к. ожидание происходит внутри функции API. Чтобы оптимизировать процесс ожидания, часто вместо функции `GetMessage` используется функция `PeekMessage`. Данная функция не ждет сообщения в очереди сообщений, а сразу возвращает управление. Таким образом, процесс ожидания может регулироваться внутри приложения. Рассмотрим другую структуру цикла обработки сообщений:

```
MSG_LOOP:
        PUSH    PM_NOREMOVE
        PUSH    0
        PUSH    0
        PUSH    0
        PUSH    OFFSET MSG
;проверить очередь команд
        CALL    PeekMessageA@20
        CMP     EAX,0
        JZ      NO_MES
        PUSH    0
        PUSH    0
        PUSH    0
        PUSH    OFFSET MSG
;взять сообщение из очереди команд
        CALL    GetMessageA@16
        CMP     AX, 0
        JZ      END_LOOP
        PUSH    OFFSET MSG
        CALL    TranslateMessage@4
        PUSH    OFFSET MSG
        CALL    DispatchMessageA@4
        JMP     MSG_LOOP
NO_MES:
;во время простоя выполнить дополнительный набор команд
        ...
        JMP     MSG_LOOP
END_LOOP:
```

Обратите внимание на функцию `PeekMesssage`. Она имеет такой же набор первых четырех параметров, как и функция `GetMessage`. Пятый параметр задает режимы выполнения этой функции. В данном случае мы используем константу `PM_NOREMOVE`, которая предупреждает функцию, что она не должна уда-

лять сообщение из очереди. При этом функция возвращает 0, если сообщений в очереди нет. Если же сообщения в очереди существуют, то мы достаем их оттуда при помощи опять-таки функции `GetMessage`. При этом у программы появляется "свободное время", чтобы выполнить еще какую-нибудь работу. Это очень полезный прием, который часто используют программы, выполняющие какую-либо графическую анимацию. Данный подход более эффективен, чем использование сообщений таймера.

Передача параметров через стек

Здесь мне хотелось бы рассмотреть подробнее вопрос о передаче параметров через стек. Это не единственный способ передачи параметров, но именно через стек передаются параметры API-функциям, поэтому на это необходимо обратить внимание особо. Состояние стека до и после вызова процедуры приводится на рис. 1.2.2. Данный рисунок демонстрирует стандартный вход в процедуру, практикующийся в таких языках высокого уровня, как Паскаль и Си. При входе в процедуру выполняется стандартная последовательность команд:

```
PUSH EBP
MOV EBP, ESP
SUB ESP, N   ; N — количество байтов для локальных переменных
```

Адрес первого параметра определяется как `[EBP+8H]`, что мы уже неоднократно использовали. Адрес первой локальной переменной, если она зарезервирована, определяется как `[EBP-4]` (имеется в виду переменная типа `DWORD`). На ассемблере не очень удобно использовать локальные переменные, и мы не будем резервировать для них место *(см. главу 2.7)*. В конце процедуры идут команды:

```
MOV ESP, EBP
POP EBP
RET M
```

Здесь `M` — количество байтов, взятых у стека для передачи параметров.

Такого же результата можно добиться, используя команду

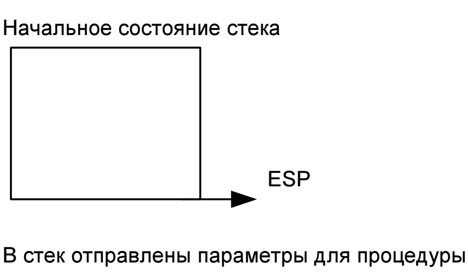
```
ENTER N,0
(PUSH EBP\MOV EBP, ESP\SUB ESP)
```

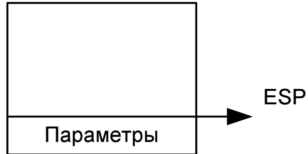
в начале процедуры и

```
LEAVE
(MOV ESP, EBP\POP EBP)
```

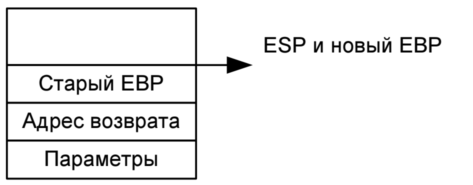
в конце процедуры. Эти команды появились еще у 286-ого процессора и дали возможность несколько оптимизировать транслируемый код программы,

особенно в тех случаях, когда речь идет о больших по объему модулях, создаваемых на языке высокого уровня.









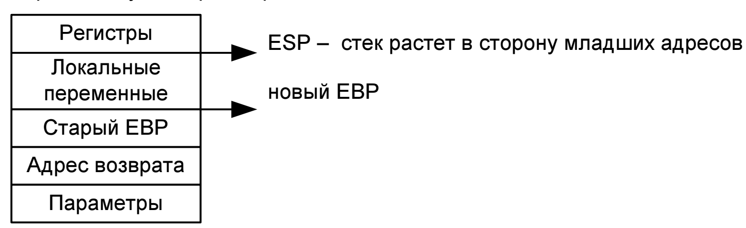
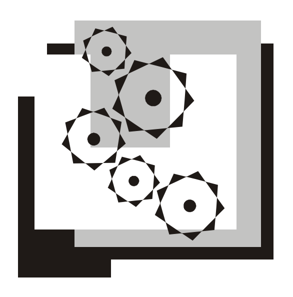


Рис. 1.2.2. Схема передачи параметров в процедуру
(стек растет в сторону младших адресов)

Хотелось бы остановиться еще на одном вопросе, который связан со структурой процедуры и ее вызова. Существуют два наиболее часто встречающихся подхода (см. [1]) к передаче параметров, или, как еще говорят, соглашения (конвенции) о передаче параметров. Условно первый подход принято называть С-подходом, а второй — Паскаль-подходом. Первый подход предполагает, что процедура "не знает", сколько параметров находится в стеке. Естественно, в этом случае освобождение стека от параметров должно происходить после команды вызова процедуры, например, с помощью команды `POP` или команды `ADD ESP,N` (`N` — количество байтов, занятых параметрами). Второй подход основан на том, что количество параметров фиксировано, поэтому стек можно освободить в самой процедуре. Это достигается за счет выполнения команды `RET N` (`N` — количество байтов в параметрах). Как вы уже, наверное, догадались, вызов функций API осуществляется по второй схеме. Впрочем, есть и исключения, о которых вы узнаете несколько позже *(*см. описание функции `wsprintfA` в *главе 2.3*).



Глава 1.3

Примеры простых программ на ассемблере

Данная глава целиком посвящена примерам, демонстрирующим технику создания окон и других элементов управления, основы которой были изложены в предыдущей главе.

Принципы построения оконных приложений

По обыкновению, я попробую сформулировать несколько положений, которые помогут в дальнейшем довольно легко манипулировать окнами и создавать гибкие, мощные, производительные приложения.

- Свойства конкретного окна задаются при вызове функции `CreateWindowEx` определением параметра `Style`. Константы, определяющие свойства окна, содержатся в специальных файлах, которые подключаются при компиляции. Поскольку свойства фактически определяются значением того или иного бита в константе, комбинация свойств — это просто сумма (или команда `OR`) битовых констант. В отличие от многих рекомендаций для разработчиков, все константы в моей книге определяются непосредственно в программах. Это вещь еще и психологическая — ваша программа становиться самодостаточной.
- Окно создается на основе зарегистрированного класса окон. Окно может содержать элементы управления — кнопки, поля редактирования, списки, полосы прокрутки и т. д. Все эти элементы могут создаваться как окна с предопределенными классами (для кнопок — `BUTTON`, для окна редактирования — `EDIT`, для списка — `LISTBOX`, для комбинированного списка — `COMBOBOX` и т. д.).
- Система общается с окном, а следовательно, и с самим приложением посредством посылки сообщений. Эти сообщения должны обрабатываться

процедурой окна. Программирование под Windows в значительной степени является программированием обработчиков таких сообщений. Сообщения генерируются системой также в случаях каких-либо визуальных событий, происходящих с окном или элементами[1] управления на нем. К таким событиям относятся перемещение окна или изменение его размеров, нажатия кнопок, выбор элемента в списке, перемещение курсора мыши в области окна и т. д. Это и понятно, программа должна как-то реагировать на подобные события.

- Сообщение имеет код (будем обозначать его в программе `MES`) и два параметра (`WPARAM` и `LPARAM`). Для каждого кода сообщения придумано свое макроимя, хотя это всего лишь целое число. Например, сообщение `WM_CREATE` (числовое значение 1) приходит один раз, когда окно создается, `WM_PAINT` посылается окну[2] при его перерисовке. Сообщение `WM_RBUTTONDOWN` генерируется, если щелкнуть правой кнопкой мыши при расположении курсора мыши в области окна и т. д. Параметры сообщения могут не иметь никакого смысла либо играть уточняющую роль. Например, сообщение `WM_COMMAND` генерируется системой, когда что-то происходит с элементами управления окна. В этом случае по значению параметров можно определить, какой это элемент и что с ним произошло (`LPARAM` — дескриптор элемента, старшее слово `WPARAM` — событие, младшее слово `WPARAM` — обычно идентификатор ресурса, *см. часть II*). Можно сказать, что сообщение `WM_COMMAND` — это сообщение от элемента в окне.
- Сообщение может генерироваться не только системой, но и самой программой. Например, можно послать сообщение-команду какому-либо элементу управления (добавить элемент в список, передать строку в окно редактирования и т. п.). Иногда посылка сообщений используется как прием программирования. Например, можно придумать свои сообщения так, чтобы при их посылке программа выполнила те или иные действия. Естественно, это сообщение должно "отлавливаться" либо в процедуре какого-либо окна, либо в цикле обработки сообщений. Такой подход очень удобен, поскольку позволяет фактически осуществлять циклические алгоритмы так, чтобы возможные изменения с окном во время исполнения такого цикла сразу проявлялись на экране.

---

[1] Вообще, можно выделить как управляющие элементы (кнопки, переключатели), так и управляемые элементы (окна редактирования, списки), но мы все их будем называть элементами управления.

[2] Хотя на самом деле вызывается процедура окна с соответствующими значениями параметров, мы и в дальнейшем будем говорить о посылке окну сообщения.

Окно с кнопкой

Приступаем к разбору следующего примера (листинг 1.3.1). При запуске этой программы на экране появляется окно с кнопкой **Выход**. Нажатие этой кнопки, как вы понимаете, должно привести к выходу из программы. Сама кнопка создается в момент прихода в оконную процедуру сообщения `WM_CREATE`. Обратите внимание, что сама кнопка создается API-функцией `CreateWindowEx`. Как и для любого окна, функция возвращает его дескриптор, который затем может использоваться для работы с окном.

Самое важное здесь — "отловить" нажатие кнопки. Но в действительности все просто: в начале проверка сообщения `WM_COMMAND`, а затем проверяем `LPARAM` — здесь хранится дескриптор (уникальный номер) окна (как я уже сказал, в Windows любой элемент окна, в том числе и кнопка, также рассматриваются как окна). В данном случае для кнопки этого уже достаточно, чтобы определить событие[3].

Да, и обратите внимание на свойства кнопки, которую мы создаем как окно, — это наиболее типичное сочетание свойств, но далеко не единственное. Например, если вы хотите, чтобы кнопка содержала пиктограмму, то необходимым условием для этого будет свойство `BS_ICON` (или `BS_BITMAP`).

Листинг 1.3.1. Пример окна с кнопкой выхода

```
;файл button.inc
;константы
;сообщение приходит при закрытии окна
WM_DESTROY                   equ 2
;сообщение приходит при создании окна
WM_CREATE                    equ 1
;сообщение при щелчке левой кнопкой мыши в области окна
WM_LBUTTONDOWN               equ 201h
WM_COMMAND                   equ 111h
;свойства окна
CS_VREDRAW                   equ 1h
CS_HREDRAW                   equ 2h
CS_GLOBALCLASS               equ 4000h
WS_OVERLAPPEDWINDOW          equ 000CF0000H
```

[3] Честно говоря, здесь я, дорогой читатель, немного грешен. Убедившись, что событие произошло именно с кнопкой, нам бы следовало определить, какое событие произошло, проверив старшее слово параметра `WPARAM` (событие `BN_CLICKED = 0`). Не вдаваясь в подробности, замечу, что в большинстве примеров, которые мы разбираем, для кнопки этого делать не обязательно.

```
STYLE equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
BS_DEFPUSHBUTTON            equ 1h
WS_VISIBLE                  equ 10000000h
WS_CHILD                    equ 40000000h
STYLBTN equ WS_CHILD+BS_DEFPUSHBUTTON+WS_VISIBLE
;идентификатор стандартной пиктограммы
IDI_APPLICATION            equ 32512
;идентификатор курсора
IDC_ARROW                  equ 32512
;режим показа окна - нормальный
SW_SHOWNORMAL              equ 1
;прототипы внешних процедур
EXTERN  MessageBoxA@16:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND         DD ?
      MSMESSAGE      DD ?
      MSWPARAM       DD ?
      MSLPARAM       DD ?
      MSTIME         DD ?
      MSPT           DD ?
MSGSTRUCT ENDS
;----структура класса окон
WNDCLASS STRUC
      CLSSTYLE       DD ?
      CLWNDPROC      DD ?
      CLSCBCLSEX     DD ?
      CLSCBWNDEX     DD ?
      CLSHINST       DD ?
      CLSHICON       DD ?
      CLSHCURSOR     DD ?
      CLBKGROUND     DD ?
```

```
      CLMENNAME      DD ?
      CLNAME         DD ?
WNDCLASS ENDS
;файл button.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include button.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      NEWHWND        DD 0
      MSG            MSGSTRUCT <?>
      WC             WNDCLASS  <?>
      HINST          DD 0 ;дескриптор приложения
      TITLENAME      DB 'Пример - кнопка выхода',0
      CLASSNAME      DB 'CLASS32',0
      CPBUT          DB 'Выход',0 ;выход
      CLSBUTN        DB 'BUTTON',0
      HWNDBTN        DWORD 0
      CAP            DB 'Сообщение',0
      MES            DB 'Конец работы программы',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH    0
      CALL    GetModuleHandleA@4
      MOV     HINST, EAX
REG_CLASS:
;заполнить структуру окна
;стиль
      MOV     WC.CLSSTYLE,STYLE
;процедура обработки сообщений
      MOV     WC.CLWNDPROC, OFFSET WNDPROC
      MOV     WC.CLSCBCLSEX, 0
      MOV     WC.CLSCBWNDEX, 0
      MOV     EAX, HINST
      MOV     WC.CLSHINST, EAX
;----------пиктограмма окна
      PUSH    IDI_APPLICATION
      PUSH    0
```

```
      CALL    LoadIconA@8
      MOV     WC.CLSHICON, EAX
;----------курсор окна
      PUSH    IDC_ARROW
      PUSH    0
      CALL    LoadCursorA@8
      MOV     WC.CLSHCURSOR, EAX
;----------
      MOV     WC.CLBKGROUND, 17 ;цвет окна
      MOV     DWORD PTR WC.CLMENNAME,0
      MOV     DWORD PTR WC.CLNAME,OFFSET CLASSNAME
      PUSH    OFFSET WC
      CALL    RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH    0
      PUSH    HINST
      PUSH    0
      PUSH    0
      PUSH    400     ; DY - высота окна
      PUSH    400     ; DX - ширина окна
      PUSH    100     ; Y-координата левого верхнего угла
      PUSH    100     ; X-координата левого верхнего угла
      PUSH    WS_OVERLAPPEDWINDOW
      PUSH    OFFSET TITLENAME ;имя окна
      PUSH    OFFSET CLASSNAME ;имя класса
      PUSH    0
      CALL    CreateWindowExA@48
;проверка на ошибку
      CMP     EAX,0
      JZ      _ERR
      MOV     NEWHWND, EAX ;дескриптор окна
;-----------------------------------
      PUSH    SW_SHOWNORMAL
      PUSH    NEWHWND
      CALL    ShowWindow@8    ; показать созданное окно
;-----------------------------------
      PUSH    NEWHWND
      CALL    UpdateWindow@4  ; перерисовать видимую часть окна,
                              ; сообщение WM_PAINT
;цикл обработки сообщений
MSG_LOOP:
      PUSH    0
      PUSH    0
      PUSH    0
      PUSH    OFFSET MSG
      CALL    GetMessageA@16
```

```
      CMP     EAX, 0
      JE      END_LOOP
      PUSH    OFFSET MSG
      CALL    TranslateMessage@4
      PUSH    OFFSET MSG
      CALL    DispatchMessageA@4
      JMP     MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
      PUSH    MSG.MSWPARAM
      CALL    ExitProcess@4
_ERR:
      JMP     END_LOOP
;-------------------------------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC     PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    EBX
      PUSH    ESI
      PUSH    EDI
      CMP     DWORD PTR [EBP+0CH], WM_DESTROY
      JE      WMDESTROY
      CMP     DWORD PTR [EBP+0CH], WM_CREATE
      JE      WMCREATE
      CMP     DWORD PTR [EBP+0CH], WM_COMMAND
      JE      WMCOMMND
      JMP     DEFWNDPROC
WMCOMMND:
      MOV     EAX,HWNDBTN
      CMP     DWORD PTR [EBP+14H], EAX  ; не кнопка ли нажата?
      JE      WMDESTROY
      MOV     EAX, 0
      JMP     FINISH
WMCREATE:
;создать окно-кнопку
      PUSH    0
      PUSH    [HINST]
      PUSH    0
      PUSH    DWORD PTR [EBP+08H]
      PUSH    20              ; DY
```

```
      PUSH    60             ; DX
      PUSH    10             ; Y
      PUSH    10             ; X
      PUSH    STYLBTN
      PUSH    OFFSET CPBUT   ; имя окна
      PUSH    OFFSET CLSBUTN ; имя класса
      PUSH    0
      CALL    CreateWindowExA@48
      MOV     HWNDBTN,EAX    ; запомнить дескриптор кнопки
      MOV     EAX, 0
      JMP     FINISH
DEFWNDPROC:
      PUSH    DWORD PTR [EBP+14H]
      PUSH    DWORD PTR [EBP+10H]
      PUSH    DWORD PTR [EBP+0CH]
      PUSH    DWORD PTR [EBP+08H]
      CALL    DefWindowProcA@16
      JMP     FINISH
WMDESTROY:
      PUSH    0        ;MB_OK
      PUSH    OFFSET CAP
      PUSH    OFFSET MES
      PUSH    DWORD PTR [EBP+08H] ; дескриптор окна
      CALL    MessageBoxA@16
      PUSH    0
      CALL    PostQuitMessage@4   ; сообщение WM_QUIT
      MOV     EAX, 0
FINISH:
      POP     EDI
      POP     ESI
      POP     EBX
      POP     EBP
      RET     16
WNDPROC ENDP
_TEXT ENDS
END START
```

Трансляция программы:

```
ml /c /coff prog.asm
link /subsystem:windows prog.obj
```

Окно с полем редактирования

Второй пример касается использования поля редактирования. Результат работы программы показан на рис. 1.3.1, а сама программа — в листинге 1.3.2.

При нажатии кнопки **Выход** появляется окно — сообщение с отредактированной строкой. Заметим, что и кнопка и поле редактирования создаются как обычные окна, из заранее созданных классов.

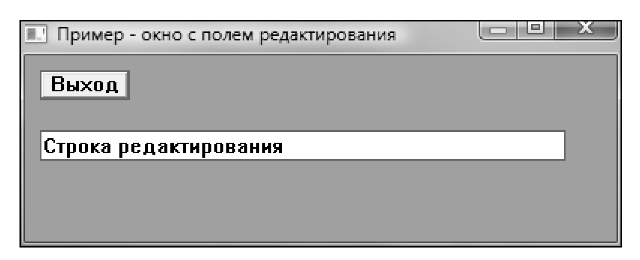


Рис. 1.3.1. Работа программы со строкой редактирования (программа из листинга 1.3.2)

Обратите внимание на то, как осуществляется посылка сообщения окну (управляющему элементу). Для этого используют в основном две функции: `SendMessage` и `PostMessage`. Отличие их друг от друга заключается в том, что первая вызывает процедуру окна с соответствующими параметрами и ждет, когда та возвратит управление; вторая функция ставит сообщение в очередь и сразу возвращает управление. Первым параметром функций является дескриптор окна, куда посылается сообщение. Вторым параметром — код сообщения. Далее идут два параметра, уточняющие код сообщения (`WPARAM` и `LPARAM`).

В программе сообщения посылаются полю редактирования. Первый раз посылается сообщение `WM_SETTEXT`, для установки там указанной строки. Второй раз посылается сообщение `WM_GETETEXT`, для того чтобы получить строку, которая находится в поле редактирования. Оба этих сообщения являются универсальными и позволяют обращаться к различным элементам управления, в том числе и к обычным окнам. Суть только в том, как интерпретировать отправляемую или получаемую строку. В случае обычного окна эта строка является просто заголовком, в случае кнопки — надписью на ней.

Листинг 1.3.2. Пример окна с полем редактирования

```
;файл edit.inc
;константы
WM_SETFOCUS          equ 7h
;сообщение приходит при закрытии окна
WM_DESTROY           equ 2
;сообщение приходит при создании окна
WM_CREATE            equ 1
;сообщение, если что-то происходит с элементами в окне
```

```
WM_COMMAND          equ 111h
;сообщение, позволяющее послать элементу строку
WM_SETTEXT          equ 0Ch
;сообщение, позволяющее получить строку
WM_GETTEXT          equ 0Dh
;свойства окна
CS_VREDRAW          equ 1h
CS_HREDRAW          equ 2h
CS_GLOBALCLASS      equ 4000h
WS_TABSTOP          equ 10000h
WS_SYSMENU          equ 80000h
WS_OVERLAPPEDWINDOW  equ 0+WS_TABSTOP+WS_SYSMENU
STYLE               equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
CS_HREDRAW          equ 2h
BS_DEFPUSHBUTTON    equ 1h
WS_VISIBLE          equ 10000000h
WS_CHILD            equ 40000000h
WS_BORDER           equ 800000h
STYLBTN   equ WS_CHILD+BS_DEFPUSHBUTTON+WS_VISIBLE+WS_TABSTOP
STYLEDT   equ WS_CHILD+WS_VISIBLE+WS_BORDER+WS_TABSTOP
;идентификатор стандартной пиктограммы
IDI_APPLICATION     equ 32512
;идентификатор курсора
IDC_ARROW           equ 32512
;режим показа окна - нормальный
SW_SHOWNORMAL       equ 1
;прототипы внешних процедур
EXTERN  SetFocus@4:NEAR
EXTERN  SendMessageA@16:NEAR
EXTERN  MessageBoxA@16:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
```

```
      MSHWND         DD ?
      MSMESSAGE      DD ?
      MSWPARAM       DD ?
      MSLPARAM       DD ?
      MSTIME         DD ?
      MSPT           DD ?
MSGSTRUCT ENDS
;----структура класса окон
WNDCLASS STRUC
      CLSSTYLE       DD ?
      CLWNDPROC      DD ?
      CLSCBCLSEX     DD ?
      CLSCBWNDEX     DD ?
      CLSHINST       DD ?
      CLSHICON       DD ?
      CLSHCURSOR     DD ?
      CLBKGROUND     DD ?
      CLMENNAME      DD ?
      CLNAME         DD ?
WNDCLASS ENDS
;файл edit.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include edit.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      NEWHWND        DD 0
      MSG            MSGSTRUCT <?>
      WC             WNDCLASS  <?>
      HINST          DD 0 ;дескриптор приложения
      TITLENAM       DB 'Пример - окно с полем редактирования',0
      CLASSNAME      DB 'CLASS32',0
      CPBUT          DB 'Выход',0 ;выход
      CPEDT          DB ' ',0
      CLSBUTN        DB 'BUTTON',0
      CLSEDIT        DB 'EDIT',0
      HWNDBTN        DD 0
      HWNDEDT        DD 0
      CAP            DB 'Сообщение',0
      MES            DB 'Конец работы программы',0
      TEXT           DB 'Строка редактирования',0
```

```
                    DB 50 DUP(0) ;продолжение буфера
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH    0
      CALL    GetModuleHandleA@4
      MOV     HINST, EAX
REG_CLASS:
;заполнить структуру окна
; стиль
      MOV     WC.CLSSTYLE,STYLE
;процедура обработки сообщений
      MOV     WC.CLWNDPROC, OFFSET WNDPROC
      MOV     WC.CLSCBCLSEX, 0
      MOV     WC.CLSCBWNDEX, 0
      MOV     EAX, HINST
      MOV     WC.CLSHINST, EAX
;----------пиктограмма окна
      PUSH    IDI_APPLICATION
      PUSH    0
      CALL    LoadIconA@8
      MOV     WC.CLSHICON, EAX
;----------курсор окна
      PUSH    IDC_ARROW
      PUSH    0
      CALL    LoadCursorA@8
      MOV     WC.CLSHCURSOR, EAX
;----------
      MOV     WC.CLBKGROUND, 17 ;цвет окна
      MOV     DWORD PTR WC.CLMENNAME,0
      MOV     DWORD PTR WC.CLNAME,OFFSET CLASSNAME
      PUSH    OFFSET WC
      CALL    RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH    0
      PUSH    HINST
      PUSH    0
      PUSH    0
      PUSH    150    ; DY - высота окна
      PUSH    400    ; DX - ширина окна
      PUSH    100    ; Y-координата левого верхнего угла
      PUSH    100    ; X-координата левого верхнего угла
      PUSH    WS_OVERLAPPEDWINDOW
      PUSH    OFFSET TITLENAME   ; имя окна
```

```
      PUSH    OFFSET CLASSNAME   ; имя класса
      PUSH    0
      CALL    CreateWindowExA@48
;проверка на ошибку
      CMP    EAX,0
      JZ     _ERR
      MOV    NEWHWND, EAX   ; дескриптор окна
;----------------------------------
      PUSH    SW_SHOWNORMAL
      PUSH    NEWHWND
      CALL    ShowWindow@8  ; показать созданное окно
;----------------------------------
      PUSH    NEWHWND
      CALL    UpdateWindow@4 ;команда перерисовать видимую
                             ;часть окна, сообщение WM_PAINT
;цикл обработки сообщений
MSG_LOOP:
      PUSH    0
      PUSH    0
      PUSH    0
      PUSH    OFFSET MSG
      CALL    GetMessageA@16
      CMP     EAX, 0
      JE      END_LOOP
      PUSH    OFFSET MSG
      CALL    TranslateMessage@4
      PUSH    OFFSET MSG
      CALL    DispatchMessageA@4
      JMP     MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
      PUSH    MSG.MSWPARAM
      CALL    ExitProcess@4
_ERR:
      JMP     END_LOOP
;----------------------------------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    EBX
```

```
      PUSH    ESI
      PUSH    EDI
      CMP     DWORD PTR [EBP+0CH],WM_DESTROY
      JE      WMDESTROY
      CMP     DWORD PTR [EBP+0CH],WM_CREATE
      JE      WMCREATE
      CMP     DWORD PTR [EBP+0CH],WM_COMMAND
      JE      WMCOMMND
      JMP     DEFWNDPROC
WMCOMMND:
      MOV     EAX,HWNDBTN
      CMP     DWORD PTR [EBP+14H],EAX
      JNE     NODESTROY
;получить отредактированную строку
      PUSH    OFFSET TEXT
      PUSH    150
      PUSH    WM_GETTEXT
      PUSH    HWNDEDT
      CALL    SendMessageA@16
;показать эту строку
      PUSH    0
      PUSH    OFFSET CAP
      PUSH    OFFSET TEXT
      PUSH    DWORD PTR [EBP+08H] ;дескриптор окна
      CALL    MessageBoxA@16
;на выход
      JMP     WMDESTROY
NODESTROY:
      MOV     EAX, 0
      JMP     FINISH
WMCREATE:
;создать окно-кнопку
      PUSH    0
      PUSH    HINST
      PUSH    0
      PUSH    DWORD PTR [EBP+08H]
      PUSH    20      ; DY
      PUSH    60      ; DX
      PUSH    10      ; Y
      PUSH    10      ; X
      PUSH    STYLBTN
      PUSH    OFFSET CPBUT      ; имя окна
      PUSH    OFFSET CLSBUTN    ; имя класса
      PUSH    0
      CALL    CreateWindowExA@48
      MOV     HWNDBTN,EAX       ; запомнить дескриптор кнопки
```

```
;создать поле редактирования
      PUSH    0
      PUSH    HINST
      PUSH    0
      PUSH    DWORD PTR [EBP+08H]
      PUSH    20      ; DY
      PUSH    350     ; DX
      PUSH    50      ; Y
      PUSH    10      ; X
      PUSH    STYLEDT
      PUSH    OFFSET CPEDT   ;имя окна
      PUSH    OFFSET CLSEDIT ;имя класса
      PUSH    0
      CALL    CreateWindowExA@48
      MOV     HWNDEDT,EAX
;---------поместить строку в окно редактирования
      PUSH    OFFSET TEXT
      PUSH    0
      PUSH    WM_SETTEXT
      PUSH    HWNDEDT
      CALL    SendMessageA@16
;---------установить фокус на окне редактирования
      PUSH    HWNDEDT
      CALL    SetFocus@4
;-----------------------------------------------
      MOV     EAX, 0
      JMP     FINISH
DEFWNDPROC:
      PUSH    DWORD PTR [EBP+14H]
      PUSH    DWORD PTR [EBP+10H]
      PUSH    DWORD PTR [EBP+0CH]
      PUSH    DWORD PTR [EBP+08H]
      CALL    DefWindowProcA@16
      JMP     FINISH
WMDESTROY:
      PUSH    0          ;MB_OK
      PUSH    OFFSET CAP
      PUSH    OFFSET MES
      PUSH    DWORD PTR [EBP+08H] ; дескриптор окна
      CALL    MessageBoxA@16
      PUSH    0
      CALL    PostQuitMessage@4   ; сообщение WM_QUIT
      MOV     EAX, 0
FINISH:
      POP     EDI
      POP     ESI
```

```
        POP     EBX
        POP     EBP
        RET     16
WNDPROC ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 1.3.2:

```
ml /c /coff prog.asm
link /subsystem:windows prog.obj
```

Окно со списком

Рассмотрим более сложный пример — окно-список (см. рис. 1.3.2). При создании окна в список помещаются названия цветов. Если произвести двойной щелчок по цвету, то появится окно-сообщение с названием этого цвета.

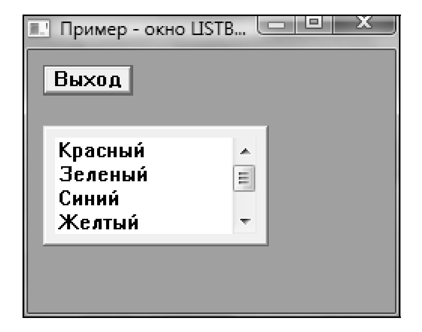


Рис. 1.3.2. Пример окна со списком

Двойной щелчок по элементу списка определяется по следующей схеме: отслеживается событие, происходящее со списком, а далее по старшему слову параметра `WPARAM` определяется, какое событие имело место (параметр `[EBP+10H]`, а его старшая часть — `[EBP+12H]`). Как видим, обработка осуществляется в три этапа:

1. Перехват сообщения `WM_COMMAND`.
2. Определение, от какого элемента пришло сообщение.
3. Определение самого события.

Параметр `LPARAM` содержит в младшем слове код события (в нашем случае это `LBN_DBLCLK`). Старшее слово параметра, как вы понимаете, содержит дескриптор элемента управления. Но обработка сообщения от списка этим еще не

заканчивается, так требуется еще определить, от какого элемента списка пришло сообщение. При помощи сообщения `LB_GETCURSEL` можно получить индекс текущего элемента. Причем индекс возвращается в регистре `EAX` (т. е. самой функцией `SendMessage`). И, наконец, используя сообщение `LB_GETTEXT`, мы можем получить значение самого элемента списка.

Не имея возможности подробно останавливаться на различных свойствах элементов управления, укажу основную идею такого элемента, как список. Каждый элемент списка имеет следующие атрибуты, по которым он может быть найден среди других элементов списка: порядковый номер элемента в списке, название элемента (строка), уникальный номер элемента в списке. Последняя характеристика наиболее важна, т. к. позволяет однозначно идентифицировать элемент, порядковый номер которого и название может меняться в процессе работы со списком.

Обратите внимание, как заполняется список. Мы заранее приготовили массив строк, а по адресу `PS` расположили адреса этих строк (`PS` — адрес первой строки, `PS+4` — адрес второй строки и т. д.). Далее используется все та же функция `SendMessage`, с помощью которой списку посылается сообщение `LB_ADDSTRING` (`LB` — это ListBox), что приводит к добавлению к списку нового элемента.

Итак, код представлен в листинге 1.3.3.

Листинг 1.3.3. Пример окна с простым списком

```
;файл list.inc
;константы
WM_SETFOCUS          equ 7h
;сообщение приходит при закрытии окна
WM_DESTROY           equ 2
;сообщение приходит при создании окна
WM_CREATE            equ 1
;сообщение, если что-то происходит с элементами
;на окне
WM_COMMAND           equ 111h
;сообщение, позволяющее послать элементу строку
WM_SETTEXT           equ 0Ch
;сообщение, позволяющее получить строку
WM_GETTEXT           equ 0Dh
;сообщение - команда добавить строку
LB_ADDSTRING         equ 180h
LB_GETTEXT           equ 189h
LB_GETCURSEL         equ 188h
LBN_DBLCLK           equ 2
```

```
;свойства окна
CS_VREDRAW          equ 1h
CS_HREDRAW          equ 2h
CS_GLOBALCLASS      equ 4000h
WS_TABSTOP          equ 10000h
WS_SYSMENU          equ 80000h
WS_THICKFRAME       equ 40000h
WS_OVERLAPPEDWINDOW equ WS_TABSTOP+WS_SYSMENU
STYLE equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
CS_HREDRAW          equ 2h
BS_DEFPUSHBUTTON    equ 1h
WS_VISIBLE          equ 10000000h
WS_CHILD            equ 40000000h
WS_BORDER           equ 800000h
WS_VSCROLL          equ 200000h
LBS_NOTIFY          equ 1h
STYLBTN equ WS_CHILD+BS_DEFPUSHBUTTON+WS_VISIBLE+WS_TABSTOP
STYLLST             equ
WS_THICKFRAME+WS_CHILD+WS_VISIBLE+WS_BORDER+WS_TABSTOP+WS_VSCROLL+LBS_NOTIFY
;идентификатор стандартной пиктограммы
IDI_APPLICATION     equ 32512
;идентификатор курсора
IDC_ARROW           equ 32512
;режим показа окна - нормальный
SW_SHOWNORMAL       equ 1
;прототипы внешних процедур
EXTERN  SetFocus@4:NEAR
EXTERN  SendMessageA@16:NEAR
EXTERN  MessageBoxA@16:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND        DD ?
```

```
      MSMESSAGE      DD ?
      MSWPARAM       DD ?
      MSLPARAM       DD ?
      MSTIME         DD ?
      MSPT           DD ?
MSGSTRUCT ENDS
;----структура класса окон
WNDCLASS STRUC
      CLSSTYLE       DD ?
      CLWNDPROC      DD ?
      CLSCBCLSEX     DD ?
      CLSCBWNDEX     DD ?
      CLSHINST       DD ?
      CLSHICON       DD ?
      CLSHCURSOR     DD ?
      CLBKGROUND     DD ?
      CLMENNAME      DD ?
      CLNAME         DD ?
WNDCLASS ENDS
;файл list.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include list.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      NEWHWND        DD 0
      MSG            MSGSTRUCT <?>
      WC             WNDCLASS  <?>
      HINST          DD 0 ;дескриптор приложения
      TITLENAME      DB 'Пример - окно LISTBOX',0
      CLASSNAME      DB 'CLASS32',0
      CPBUT          DB 'Выход',0 ;выход
      CPLST          DB ' ',0
      CLSBUTN        DB 'BUTTON',0
      CLSLIST        DB 'LISTBOX',0
      HWNDBTN        DWORD 0
      HWNDLST        DWORD 0
      CAP            DB 'Сообщение',0
      CAP1           DB 'Выбран',0
      MES            DB 'Конец работы программы',0
```

```
;массив строк
      STR1    DB 'Красный',0
      STR2    DB 'Зеленый',0
      STR3    DB 'Синий',0
      STR4    DB 'Желтый',0
      STR5    DB 'Черный',0
      STR6    DB 'Белый',0
;указатели на строки
      PS      DWORD OFFSET STR1
              DWORD OFFSET STR2
              DWORD OFFSET STR3
              DWORD OFFSET STR4
              DWORD OFFSET STR5
              DWORD OFFSET STR6
      BUF     DB 30 dup(0)
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH    0
      CALL    GetModuleHandleA@4
      MOV     HINST, EAX
REG_CLASS:
;заполнить структуру окна
;стиль
      MOV     WC.CLSSTYLE,STYLE
;процедура обработки сообщений
      MOV     WC.CLWNDPROC, OFFSET WNDPROC
      MOV     WC.CLSCBCLSEX, 0
      MOV     WC.CLSCBWNDEX, 0
      MOV     EAX, HINST
      MOV     WC.CLSHINST, EAX
;----------пиктограмма окна
      PUSH    IDI_APPLICATION
      PUSH    0
      CALL    LoadIconA@8
      MOV     WC.CLSHICON, EAX
;----------курсор окна
      PUSH    IDC_ARROW
      PUSH    0
      CALL    LoadCursorA@8
      MOV     WC.CLSHCURSOR, EAX
;----------
      MOV     WC.CLBKGROUND, 17 ;цвет окна
      MOV     DWORD PTR WC.CLMENNAME,0
```

```
      MOV     DWORD PTR WC.CLNAME,OFFSET CLASSNAME
      PUSH    OFFSET WC
      CALL    RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH    0
      PUSH    HINST
      PUSH    0
      PUSH    0
      PUSH    200    ; DY - высота окна
      PUSH    250    ; DX - ширина окна
      PUSH    100    ; Y-координата левого верхнего угла
      PUSH    100    ; X-координата левого верхнего угла
      PUSH    WS_OVERLAPPEDWINDOW
      PUSH    OFFSET TITLENAME ;имя окна
      PUSH    OFFSET CLASSNAME ;имя класса
      PUSH    0
      CALL    CreateWindowExA@48
;проверка на ошибку
      CMP     EAX,0
      JZ      _ERR
      MOV     NEWHWND, EAX ;дескриптор окна
;---------------------------------
      PUSH    SW_SHOWNORMAL
      PUSH    NEWHWND
      CALL    ShowWindow@8 ;показать созданное окно
;---------------------------------
      PUSH    NEWHWND
      CALL    UpdateWindow@4 ;команда перерисовать видимую
                             ;часть окна, сообщение WM_PAINT
;цикл обработки сообщений
MSG_LOOP:
      PUSH    0
      PUSH    0
      PUSH    0
      PUSH    OFFSET MSG
      CALL    GetMessageA@16
      CMP     EAX, 0
      JE      END_LOOP
      PUSH    OFFSET MSG
      CALL    TranslateMessage@4
      PUSH    OFFSET MSG
      CALL    DispatchMessageA@4
      JMP     MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
      PUSH    MSG.MSWPARAM
```

```
      CALL    ExitProcess@4
_ERR:
      JMP     END_LOOP
;-----------------------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]   ;LPARAM
; [EBP+10H]    ;WAPARAM
; [EBP+0CH]    ;MES
; [EBP+8]      ;HWND
WNDPROC PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    EBX
      PUSH    ESI
      PUSH    EDI
      CMP     DWORD PTR [EBP+0CH],WM_DESTROY
      JE      WMDESTROY
      CMP     DWORD PTR [EBP+0CH],WM_CREATE
      JE      WMCREATE
      CMP     DWORD PTR [EBP+0CH],WM_COMMAND
      JE      WMCOMMND
      JMP     DEFWNDPROC
WMCOMMND:
      MOV     EAX,HWNDBTN
      CMP     DWORD PTR [EBP+14H],EAX ;кнопка?
;на выход?
      JE      WMDESTROY
      MOV     EAX,HWNDLST
      CMP     DWORD PTR [EBP+14H],EAX ;список?
      JNE     NOLIST
;работаем со списком
      CMP     WORD PTR [EBP+12H],LBN_DBLCLK
      JNE     NOLIST
;двойной щелчок есть, теперь определить выбранную строку
;вначале индекс
      PUSH    0
      PUSH    0
      PUSH    LB_GETCURSEL
      PUSH    HWNDLST
      CALL    SendMessageA@16
;теперь сам текст
      PUSH    OFFSET BUF
      PUSH    EAX
      PUSH    LB_GETTEXT
      PUSH    HWNDLST
      CALL    SendMessageA@16
```

```
;сообщить, что выбрано
      PUSH    0
      PUSH    OFFSET CAP1
      PUSH    OFFSET BUF
      PUSH    DWORD PTR [EBP+08H]
      CALL    MessageBoxA@16
NOLIST:
      MOV     EAX, 0
      JMP     FINISH
WMCREATE:
;создать окно-кнопку
      PUSH    0
      PUSH    HINST
      PUSH    0
      PUSH    DWORD PTR [EBP+08H]
      PUSH    20      ; DY
      PUSH    60      ; DX
      PUSH    10      ; Y
      PUSH    10      ; X
      PUSH    STYLBTN
      PUSH    OFFSET CPBUT     ; имя окна
      PUSH    OFFSET CLSBUTN   ; имя класса
      PUSH    0
      CALL    CreateWindowExA@48
      MOV     HWNDBTN,EAX      ; запомнить дескриптор кнопки
;-----------------------------------------------
;создать окно LISTBOX
      PUSH    0
      PUSH    HINST
      PUSH    0
      PUSH    DWORD PTR [EBP+08H]
      PUSH    90      ; DY
      PUSH    150     ; DX
      PUSH    50      ; Y
      PUSH    10      ; X
      PUSH    STYLLST
      PUSH    OFFSET CPLST   ;имя окна
      PUSH    OFFSET CLSLIST ;имя класса
      PUSH    0
      CALL    CreateWindowExA@48
      MOV     HWNDLST,EAX
;заполнить список
      PUSH    PS
      PUSH    0
      PUSH    LB_ADDSTRING
      PUSH    HWNDLST
```

```
      CALL    SendMessageA@16
      PUSH    PS+4
      PUSH    0
      PUSH    LB_ADDSTRING
      PUSH    HWNDLST
      CALL    SendMessageA@16
      PUSH    PS+8
      PUSH    0
      PUSH    LB_ADDSTRING
      PUSH    HWNDLST
      CALL    SendMessageA@16
      PUSH    PS+12
      PUSH    0
      PUSH    LB_ADDSTRING
      PUSH    HWNDLST
      CALL    SendMessageA@16
      PUSH    PS+16
      PUSH    0
      PUSH    LB_ADDSTRING
      PUSH    HWNDLST
      CALL    SendMessageA@16
      PUSH    PS+20
      PUSH    0
      PUSH    LB_ADDSTRING
      PUSH    HWNDLST
      CALL    SendMessageA@16
      MOV     EAX, 0
      JMP     FINISH
DEFWNDPROC:
      PUSH    DWORD PTR [EBP+14H]
      PUSH    DWORD PTR [EBP+10H]
      PUSH    DWORD PTR [EBP+0CH]
      PUSH    DWORD PTR [EBP+08H]
      CALL    DefWindowProcA@16
      JMP     FINISH
WMDESTROY:
      PUSH    0          ;MB_OK
      PUSH    OFFSET CAP
      PUSH    OFFSET MES
      PUSH    DWORD PTR [EBP+08H]    ; дескриптор окна
      CALL    MessageBoxA@16
      PUSH    0
      CALL    PostQuitMessage@4      ; сообщение WM_QUIT
      MOV     EAX, 0
FINISH:
      POP     EDI
```

```
        POP     ESI
        POP     EBX
        POP     EBP
        RET     16
WNDPROC     ENDP
_TEXT ENDS
END START
```

Трансляция программы:

```
ml /c /coff prog.asm
link /subsystem:windows prog.obj
```

Мне бы хотелось также поговорить вот на какую тему. С окном на экране могут происходить самые интересные и необычные вещи. Например, оно по желанию пользователя может перемещаться, менять свой размер, сворачиваться и разворачиваться, наконец, другие окна могут заслонять данное окно. В таких ситуациях система посылает окну сообщение `WM_PAINT`. Такое же сообщение посылается окну при выполнении некоторых функций, связанных с перерисовкой окна, таких, например, как `UpdateWindow`. Если сообщение `WM_PAINT` не обрабатывается процедурой окна, а возвращается системе посредством функции `DefWindowProc`, то система берет на себя не только перерисовку окна (что она и так делает), но и перерисовку содержимого окна. К содержимому окна, однако, относятся только дочерние окна, которыми являются кнопки, списки, поля редактирования и другие элементы управления *(см. следующий раздел)*. В следующих главах мы, в частности, будем говорить о том, как выводить в окно текстовую и графическую информацию. Здесь, чтобы такая информация сохранялась в окне, нам не обойтись без обработки сообщения `WM_PAINT`.

Дочерние и собственные окна

Окна, как мы еще неоднократно убедимся, обладают большим количеством атрибутов. Кроме этого, окна могут находиться по отношению друг к другу в определенных отношениях. Другими словами, каждое окно может иметь родительское окно и несколько дочерних окон. Окно, не имеющее родительского окна, *называется окном верхнего уровня*. Все вышеприведенные примеры демонстрировали как раз окна верхнего уровня[4]. Элементы же, располагающиеся в окне (кнопки, списки и т. п.), являлись дочерними окнами. Кроме дочерних окон имеются и так называемые *собственные окна*. Собственное окно имеет родителя, но не является по отношению к нему дочерним

[4] Все главные окна являются окнами верхнего уровня. Обратное, разумеется, не верно.

окном. Примерами собственных окон являются так называемые окна pop-up, используемые для построения различных диалоговых панелей. Основным отличием дочернего окна от собственного окна является то, что перемещение дочернего окна ограничивается областью родительского окна, а перемещение собственного окна — областью экрана. Поведение родительского окна влияет на поведение дочерних и собственных окон:

- при скрытии или отображении родительского окна, соответственно, скрываются и отображаются дочерние и собственные окна;
- при перемещении родительского окна перемещаются дочерние (но не собственные) окна;
- при уничтожении родительского окна первым делом уничтожаются все его дочерние и собственные окна.

Дочерние окна, обладающие одним родительским окном, на экране могут перекрывать друг друга. Порядок отображения таких окон называется *Z*-порядком и может регулироваться с помощью специальных API-функций (`SetWindowPos`, `DeferWindowPos`, `GetNextWindow` и т. д.). Далее мы не будем останавливаться на этом вопросе. В листинге 1.3.4 представлена программа, как раз демонстрирующая свойства дочерних и собственных окон (рис. 1.3.3). При ее запуске возникает главное окно. Щелчок мышью в области окна вызывает появление двух окон — дочернего и собственного. Вы можете легко поэкспериментировать с ними и определить их свойства.

Листинг 1.3.4. Пример программы, создающей одно главное окно и два окна — дочернее и собственное

```
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;-------------------------------------------------
;константы
;сообщение приходит при закрытии окна
WM_DESTROY          equ 2
;сообщение приходит при создании окна
WM_CREATE           equ 1
;сообщение при щелчке левой кнопкой мыши в области окна
WM_LBUTTONDOWN      equ 201h
;свойства окна
CS_VREDRAW          equ 1h
CS_HREDRAW          equ 2h
CS_GLOBALCLASS      equ 4000h
WS_OVERLAPPEDWINDOW equ 000CF0000H
WS_POPUP            equ 80000000h
```

```
WS_CHILD             equ 40000000h
STYLE equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
BS_DEFPUSHBUTTON     equ 1h
WS_VISIBLE           equ 10000000h
WS_CHILD             equ 40000000h
STYLBTN              equ WS_CHILD+BS_DEFPUSHBUTTON+WS_VISIBLE
;идентификатор стандартной пиктограммы
IDI_APPLICATION      equ 32512
;идентификатор курсора
IDC_ARROW            equ 32512
;режим показа окна - нормальный
SW_SHOWNORMAL        equ 1
;прототипы внешних процедур
EXTERN  MessageBoxA@16:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND          DD ?
      MSMESSAGE       DD ?
      MSWPARAM        DD ?
      MSLPARAM        DD ?
      MSTIME          DD ?
      MSPT            DD ?
MSGSTRUCT ENDS
;----структура класса окон
WNDCLASS STRUC
      CLSSTYLE        DD ?
      CLWNDPROC       DD ?
      CLSCBCLSEX      DD ?
      CLSCBWNDEX      DD ?
      CLSHINST        DD ?
      CLSHICON        DD ?
      CLSHCURSOR      DD ?
```

```
      CLBKGROUND     DD ?
      CLMENNAME      DD ?
      CLNAME         DD ?
WNDCLASS ENDS
;-------------------------------------------------
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      NEWHWND        DD 0
      MSG            MSGSTRUCT <?>
      WC             WNDCLASS  <?>
      HINST          DD 0 ;дескриптор приложения
      TITLENAME      DB 'Дочерние и собственные окна',0
      TITLENAMED     DB 'Дочернее окно',0
      TITLENAMEO     DB 'Собственное окно',0
      CLASSNAME      DB 'CLASS32',0
      CLASSNAMED     DB 'CLASS321',0
      CLASSNAMEO     DB 'CLASS322',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH    0
      CALL    GetModuleHandleA@4
      MOV     HINST, EAX
REG_CLASS:
;фрагмент, где осуществляется регистрация главного окна
;стиль
      MOV     WC.CLSSTYLE,STYLE
;процедура обработки сообщений
      MOV     WC.CLWNDPROC, OFFSET WNDPROC
      MOV     WC.CLSCBCLSEX, 0
      MOV     WC.CLSCBWNDEX, 0
      MOV     EAX, HINST
      MOV     WC.CLSHINST, EAX
;----------пиктограмма окна
      PUSH    IDI_APPLICATION
      PUSH    0
      CALL    LoadIconA@8
      MOV     WC.CLSHICON, EAX
;----------курсор окна
      PUSH    IDC_ARROW
```

```
      PUSH    0
      CALL    LoadCursorA@8
      MOV     WC.CLSHCURSOR, EAX
;----------регистрация основного окна
      MOV     WC.CLBKGROUND, 17 ;цвет окна
      MOV     DWORD PTR WC.CLMENNAME,0
      MOV     DWORD PTR WC.CLNAME,OFFSET CLASSNAME
      PUSH    OFFSET WC
      CALL    RegisterClassA@4
;фрагмент, где осуществляется регистрация дочернего окна
;стиль
      MOV     WC.CLSSTYLE,STYLE
;процедура обработки сообщений
      MOV     WC.CLWNDPROC, OFFSET WNDPROCD
      MOV     WC.CLSCBCLSEX, 0
      MOV     WC.CLSCBWNDEX, 0
      MOV     EAX, HINST
      MOV     WC.CLSHINST, EAX
;----------
      MOV     WC.CLBKGROUND, 2 ;цвет окна
      MOV     DWORD PTR WC.CLMENNAME,0
      MOV     DWORD PTR WC.CLNAME,OFFSET CLASSNAMED
      PUSH    OFFSET WC
      CALL    RegisterClassA@4
;фрагмент, где осуществляется регистрация собственного окна
;стиль
      MOV     WC.CLSSTYLE,STYLE
;процедура обработки сообщений
      MOV     WC.CLWNDPROC, OFFSET WNDPROCO
      MOV     WC.CLSCBCLSEX, 0
      MOV     WC.CLSCBWNDEX, 0
      MOV     EAX, HINST
      MOV     WC.CLSHINST, EAX
;----------
      MOV     WC.CLBKGROUND, 1 ;цвет окна
      MOV     DWORD PTR WC.CLMENNAME,0
      MOV     DWORD PTR WC.CLNAME,OFFSET CLASSNAMEO
      PUSH    OFFSET WC
      CALL    RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH    0
      PUSH    HINST
      PUSH    0
      PUSH    0
      PUSH    400     ; DY - высота окна
      PUSH    600     ; DX - ширина окна
```

```
        PUSH    100    ; Y-координата левого верхнего угла
        PUSH    100    ; X-координата левого верхнего угла
        PUSH    WS_OVERLAPPEDWINDOW
        PUSH    OFFSET TITLENAME ;имя окна
        PUSH    OFFSET CLASSNAME ;имя класса
        PUSH    0
        CALL    CreateWindowExA@48
;проверка на ошибку
        CMP     EAX,0
        JZ      _ERR
        MOV     NEWHWND, EAX ;дескриптор окна
;----------------------------------
        PUSH    SW_SHOWNORMAL
        PUSH    NEWHWND
        CALL    ShowWindow@8   ; показать созданное окно
;----------------------------------
        PUSH    NEWHWND
        CALL    UpdateWindow@4 ; команда перерисовать видимую
;часть окна, сообщение WM_PAINT
;цикл обработки сообщений
MSG_LOOP:
        PUSH    0
        PUSH    0
        PUSH    0
        PUSH    OFFSET MSG
        CALL    GetMessageA@16
        CMP     EAX, 0
        JE      END_LOOP
        PUSH    OFFSET MSG
        CALL    TranslateMessage@4
        PUSH    OFFSET MSG
        CALL    DispatchMessageA@4
        JMP     MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
        PUSH    MSG.MSWPARAM
        CALL    ExitProcess@4
_ERR:
        JMP     END_LOOP

;***********************************
;процедура главного окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
```

```
; [EBP+8]     ;HWND
WNDPROC     PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    EBX
      PUSH    ESI
      PUSH    EDI
      CMP     DWORD PTR [EBP+0CH], WM_DESTROY
      JE      WMDESTROY
      CMP     DWORD PTR [EBP+0CH], WM_CREATE
      JE      WMCREATE
      CMP     DWORD PTR [EBP+0CH], WM_LBUTTONDOWN
      JE      LB
      JMP     DEFWNDPROC
WMCREATE:
      MOV     EAX,0
      JMP     FINISH
LB:
;создать дочернее окно
      PUSH    0
      PUSH    HINST
      PUSH    0
      PUSH    DWORD PTR [EBP+08H]
      PUSH    200     ; DY - высота окна
      PUSH    200     ; DX - ширина окна
      PUSH    50      ; Y-координата левого верхнего угла
      PUSH    50      ; X-координата левого верхнего угла
      PUSH    WS_CHILD OR WS_VISIBLE OR WS_OVERLAPPEDWINDOW
      PUSH    OFFSET TITLENAMED ;имя окна
      PUSH    OFFSET CLASSNAMED ;имя класса
      PUSH    0
      CALL    CreateWindowExA@48
;создать собственное окно
      PUSH    0
      PUSH    HINST
      PUSH    0
      PUSH    DWORD PTR [EBP+08H]
      PUSH    200     ; DY - высота окна
      PUSH    200     ; DX - ширина окна
      PUSH    150     ; Y-координата левого верхнего угла
      PUSH    250     ; X-координата левого верхнего угла
      PUSH    WS_POPUP OR WS_VISIBLE OR WS_OVERLAPPEDWINDOW
      PUSH    OFFSET TITLENAMEO ;имя окна
      PUSH    OFFSET CLASSNAMEO ;имя класса
      PUSH    0
      CALL    CreateWindowExA@48
```

```
DEFWNDPROC:
      PUSH    DWORD PTR [EBP+14H]
      PUSH    DWORD PTR [EBP+10H]
      PUSH    DWORD PTR [EBP+0CH]
      PUSH    DWORD PTR [EBP+08H]
      CALL    DefWindowProcA@16
      JMP     FINISH
WMDESTROY:
      PUSH    0
      CALL    PostQuitMessage@4 ;сообщение WM_QUIT
      MOV     EAX, 0
FINISH:
      POP     EDI
      POP     ESI
      POP     EBX
      POP     EBP
      RET     16
WNDPROC ENDP
;************************************
;процедура дочернего окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROCD      PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    EBX
      PUSH    ESI
      PUSH    EDI
      CMP     DWORD PTR [EBP+0CH], WM_DESTROY
      JE      WMDESTROY
      CMP     DWORD PTR [EBP+0CH], WM_CREATE
      JE      WMCREATE
      JMP     DEFWNDPROC
WMCREATE:
      JMP     FINISH
DEFWNDPROC:
      PUSH    DWORD PTR [EBP+14H]
      PUSH    DWORD PTR [EBP+10H]
      PUSH    DWORD PTR [EBP+0CH]
      PUSH    DWORD PTR [EBP+08H]
      CALL    DefWindowProcA@16
      JMP     FINISH
WMDESTROY:
      MOV     EAX, 0
```

```
FINISH:
      POP     EDI
      POP     ESI
      POP     EBX
      POP     EBP
      RET     16
WNDPROCD   ENDP

;************************************
;процедура собственного окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROCO    PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    EBX
      PUSH    ESI
      PUSH    EDI
      CMP     DWORD PTR [EBP+0CH], WM_DESTROY
      JE      WMDESTROY
      CMP     DWORD PTR [EBP+0CH], WM_CREATE
      JE      WMCREATE
      JMP     DEFWNDPROC
WMCREATE:
      JMP     FINISH
DEFWNDPROC:
      PUSH    DWORD PTR [EBP+14H]
      PUSH    DWORD PTR [EBP+10H]
      PUSH    DWORD PTR [EBP+0CH]
      PUSH    DWORD PTR [EBP+08H]
      CALL    DefWindowProcA@16
      JMP     FINISH
WMDESTROY:
      MOV     EAX, 0
FINISH:
      POP     EDI
      POP     ESI
      POP     EBX
      POP     EBP
      RET     16
WNDPROCO   ENDP
_TEXT ENDS
END START
```

Трансляция программы:

```
ml /c /coff prog.asm
link /subsystem:windows prog.obj
```

Обращаю ваше внимание на один нетривиальный момент. В программе имеются три процедуры окна, и в каждой присутствуют метки, имеющие одинаковые названия. Транслятор MASM32 автоматически считает все метки процедуры локальными, т. е. при трансляции расширяет их имена до уникальных. Таким образом, внутри процедуры мы можем свободно пользоваться переходами, не боясь, что переход будет осуществлен в другую процедуру. Подробнее о локальных метках будет сказано в *главе 2.7*.

Обратите внимание, что для каждого из трех появляющихся окон нами определена своя процедура обработки сообщений. Эти процедуры обработки сообщений не содержат команды `PostQuitMessage`. Это и понятно: закрытие дочернего или собственного окна не должно вызывать выход из программы. Во всем остальном содержимое процедуры этих окон может быть таким же, как и для главного окна (окна верхнего уровня). Они могут содержать элементы управления, иметь заголовки, обрабатывать любые сообщения и т. д.

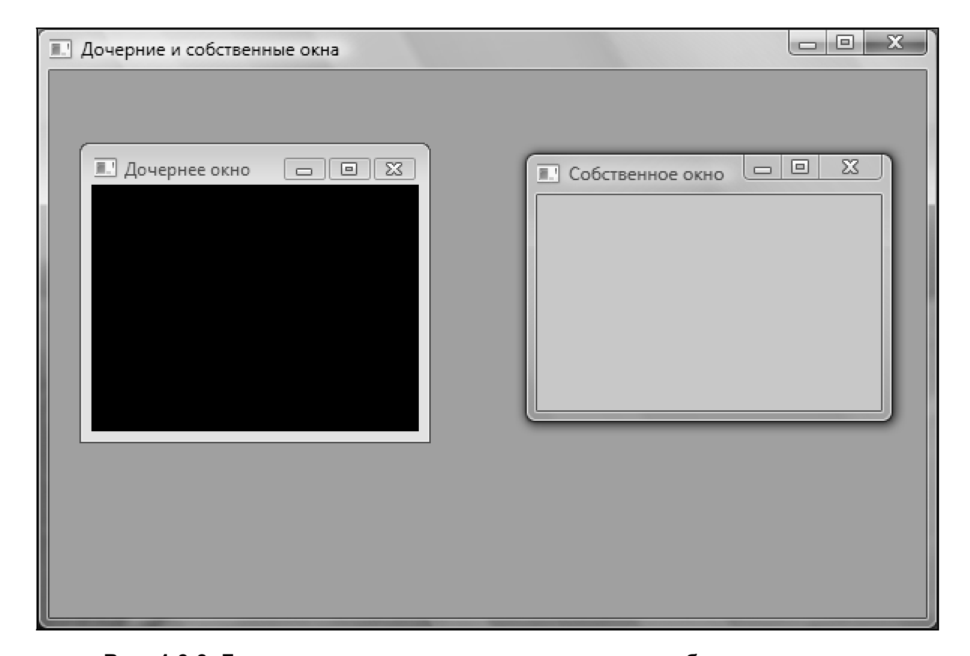


Рис. 1.3.3. Главное окно с одним дочерним и одним собственным окном

Глава 1.4

Ассемблер MASM

В данной главе мы поговорим о трансляторе с языка ассемблера MASM. Когда в конце 80-х годов XX в. я впервые "пересел" на "айбиэмку", первый вопрос, который я задал знающим людям, был об ассемблере. До этого я программировал на разных компьютерах, в основном имеющих весьма ограниченные ресурсы. Естественно, что основным языком на таких компьютерах был ассемблер[1]. Мне дали MASM, кажется, это была вторая версия. Удивительно, но тогда на ассемблере я начал писать что-то типа баз данных, что вызвало несказанное удивление окружающих. Проект мой не был закончен, но к ассемблеру я прикипел основательно. Потом мне попался Turbo Assembler версии 1.0. Он работал гораздо быстрее MASM. В дальнейшем мне приходилось использовать то один, то другой ассемблер. Как вы, наверное, уже поняли, первая любовь оказалась сильнее, к тому же Turbo Assembler теперь не поддерживается фирмой Borland.

Командная строка ML.EXE

Начнем со справочной информации о параметрах командной строки ML.EXE. Ключи командной строки представлены в табл. 1.4.1.

---

[1] Одно время в образовании был широко распространен персональный компьютер Yamaha, ОЗУ которого составляла всего 64 Кбайт (потом 128 Кбайт). Писать для такого компьютера, скажем, на языке Паскаль было, естественно, непозволительной роскошью.

Таблица 1.4.1. *Параметры командной строки программы ML.EXE*

Параметр	Комментарий
/?	Вывод помощи
/AT	Создать файл в формате COM. Для программирования в Windows этот ключ, естественно, бесполезен
/Bl<*linker*>	Использовать альтернативный компоновщик. Предполагается автоматический запуск компоновщика
/c	Компиляция без компоновки
/Cp	Сохранение регистров пользовательских идентификаторов. Может использоваться для дополнительного контроля
/Cu	Приведение всех пользовательских идентификаторов к верхнему регистру
/Cx	Сохранение регистров пользовательских идентификаторов, объявленных PUBLIC и EXTERNAL
/coff	Создание объектных файлов в формате COFF. Применение обязательно
/D<*name*>=[*string*]	Задание текстового макроса. Очень удобен для отладки с использованием условной компиляции
/EP	Листинг — текст программы с включаемыми файлами
/F<*hex*>	Размер стека в байтах. Размер стека по умолчанию равен 1 Мбайт
/Fe<*file*>	Имя исполняемого файла. Имеет смысл без параметра /c
/Fl<*file*>	Создать файл листинга
/Fm<*file*>	Создать map-файл. Имеет смысл без опции /c
/Fo<*file*>	Задать имя объектного файла
/Fpi	Включение кода эмулятора сопроцессора. Начиная с 486-го микропроцессора, данный параметр потерял актуальность
/Fr	Включить ограниченную информацию браузера
/FR	Включить полную информацию браузера
/G<c|d|z>	Использовать соглашение вызова Паскаль, C, Stdcall
/H<*number*>	Установить максимальную длину внешних имен
/I<*name*>	Добавить путь для inc-файлов. Допускается до 10 опций /I
/link<*opt*>	Опции командной строки компоновщика. Имеет смысл без опции /c
/nologo	Не показывать заголовочный текст компилятора

Таблица 1.4.1 *(окончание)*

Параметр	**Комментарий**
/Sa	Листинг максимального формата
/Sc	Включить в листинг синхронизацию
/Sf	Листинг первого прохода
/Sl<*number*>	Длина строки листинга
/Sn	Не включать в листинг таблицу символов
/Sp<number>	Высота страницы листинга
/Ss<*string*>	Текст подзаголовка листинга
/St<*string*>	Текст заголовка листинга
/Sx	Включить в листинг фрагменты условной компиляции
/Ta<*file*>	Для компилирования файлов, расширение которых отлично от asm
/W<*number*>	Устанавливает перечень событий компиляции, трактуемые как предупреждения
/WX	Трактовать предупреждения как ошибки
/w	То же, что /W0 /WX
/X	Игнорировать путь, установленный переменной окружения INCLUDE
/Zd	Отладочная информация состоит только из номеров строк
/Zf	Объявить все имена PUBLIC
/Zi	Включить полную отладочную информацию
/Zm	Включить совместимость с MASM 5.01
/Zp<n>	Установить выравнивание структур
/Zs	Выполнять только проверку синтаксиса

Запуск трансляции в MASM32 можно осуществлять с помощью специального командного файла. Это обычный текстовый файл, в котором перечислены опции запуска. Например, вместо командной строки MASM32 `ml /c mt.asm` можно создать текстовый файл mt.cmd со следующим содержанием:

```
/ml
/c
Mt.asm
```

и выполнить команду `ml @mt.cmd`.

Командная строка LINK.EXE

Перечень опций программы LINK.EXE (32 bit) и их описание приведены в табл. 1.4.2.

Таблица 1.4.2. *Опции командной строки программы LINK.EXE*

Параметр	Комментарий
`/ALIGN:`*`number`*	Определяет выравнивание секций в линейной модели. По умолчанию 4096
`/BASE:{`*`address`*`|@`*`filename`*`,`*`key`*`}`	Определяет базовый адрес (адрес загрузки). По умолчанию для exe-программы адрес 0x400000, для dll — 0x10000000
`/COMMENT: ["]`*`comment`*`["]`	Определяет комментарий, помещаемый в заголовок exe- и dll-файлов
`/DEBUG`	Создает отладочную информацию для exe- и dll-файлов. Отладочная информация помещается в pdb-файл
`/DEBUGTYPE:{CV|COFF|BOTH}`	`CV` — отладочная информация в формате Microsoft. `COFF` — отладочная информация в формате COFF (Common Object File Format). `BOTH` — создаются оба вида отладочной информации
`/DEF:`*`filename`*	Определяет def-файл
`/DEFAULTLIB:`*`library`*	Добавляет одну библиотеку к списку используемых библиотек
`/DLL`	Создать dll-файл
`/DRIVER[:{UPONLY|WDM}]`	Используется для создания NT-драйвера (kernel mode driver)
`/DUMP`	Получение дампа двоичного файла (см. листинг 1.1.11 и комментарий к нему)
`/ENTRY:`*`symbol`*	Определяет стартовый адрес (имя символьной метки) для exe- и dll-файлов
`/EXETYPE:DYNAMIC`	Данная опция используется при создании vxd-драйвера
`/EXPORT:`*`entryname`*`[=`*`internalname`*`][,@`*`ordinal`*`[,NONAME]][,DATA]`	Данная опция позволяет экспортировать функцию из вашей программы так, чтобы она была доступна для других программ. При этом создается import-библиотека

Таблица 1.4.2 (продолжение)

Параметр	**Комментарий**
`/FIXED[:NO]`	Данная опция фиксирует базовый адрес, определенный в опции `/BASE`
`/FORCE[:{MULTIPLE|UNRESOLVED}]`	Позволяет создавать исполняемый файл, даже если не найдено внешнее имя или имеется несколько разных определений
`/GPSIZE:`*`number`*	Определяет размер общих переменных для MIPS- и Alpha-платформ
`/HEAP:reserve[,`*`commit`*`]`	Определяет размер кучи (heap) в байтах. По умолчанию этот размер равен одному мегабайту
`/IMPLIB:`*`filename`*	Определяет имя import-библиотеки, если она создается
`/INCLUDE:`*`symbol`*	Добавляет имя к таблице имен, используемых отладчиком
`/INCREMENTAL:{YES|NO}`	Если установлена опция `/INCREMENTAL:YES`, то в exe-файл добавляется дополнительная информация, позволяющая быстрее перекомпилировать этот файл. По умолчанию эта информация не добавляется
`/LARGEADDRESSAWARE[:NO]`	Указывает, что приложение оперирует адресами, большими 2 Гбайт
`/LIB`	Опция перехода на управление библиотекой (см. листинг 1.1.6 и комментарий к нему)
`/LIBPATH:dir`	Определяет библиотеку, которая в первую очередь разыскивается компоновщиком
`/MACHINE:{ALPHA|ARM|IX86|MIPS|MIPS16|MIPSR41XX|PPC|SH3|SH4}`	Определяет платформу. В большинстве случаев это делать не приходится
`/MAP[:`*`filename`*`]`	Дает команду создания map-файла
`/MAPINFO:{EXPORTS|FIXUPS|LINES}`	Указывает компоновщику включить соответствующую информацию в map-файл
`/MERGE:from=to`	Объединить секцию `From` с секцией `to` и присвоить имя `to`
`/NODEFAULTLIB[:`*`library`*`]`	Игнорирует все или конкретную библиотеку
`/NOENTRY`	Необходимо для создания dll-файла

Таблица 1.4.2 (продолжение)

Параметр	Комментарий
`/NOLOGO`	Не выводить начальное сообщение компоновщика
`/OPT:{ICF[,iterations]|NOICF|NOREF|NOWIN98|REF|WIN98}`	Определяет способ оптимизации, которую выполняет компоновщик
`/ORDER:@filename`	Оптимизация программы путем вставки определенных инициализированных данных (COMDAT)
`/OUT:filename`	Определяет выходной файл
`/PDB:{`*`filename`*`|NONE}`	Определяет имя файла, содержащего информацию для отладки
`/PDBTYPE:{CON[SOLIDATE]|SEPT[YPES]}`	Определяет тип pdb-файла
`/PROFILE`	Используется для работы с профайлером (анализатором работы программы)
`/RELEASE`	Помещает контрольную сумму в выходной файл
`/SECTION:`*`name`*`,[E][R][W][S][D][K][L][P][X]`	Данная опция позволяет изменить атрибут секции
`/STACK:reserve[,`*`commit`*`]`	Определяет размер выделяемого стека. *`commit`* определяет размер памяти, интерпретируемый операционной системой
`/STUB:`*`filename`*	Определяет stub-файл, запускающийся в системе MS-DOS
`/SUBSYSTEM:{NATIVE|WINDOWS|CONSOLE|WINDOWSCE|POSIX}[,#[.##]]`	Определяет, как запускать exe-файл. `CONSOLE` — консольное приложение, `WINDOWS` — обычные графические оконные Windows-приложения, `NATIVE` — приложение для Windows NT (драйвер режима ядра), `POSIX` — создает приложение в POSIX-подсистеме Windows NT
`/SWAPRUN:{CD|NET}`	Сообщает операционной системе скопировать выходной файл в swap-файл (Windows NT)
`/VERBOSE[:LIB]`	Заставляет выводить информацию о процессе компоновки
`/VERSION:#[.#]`	Помещает информацию о версии в exe-заголовок
`/VXD`	Создать vxd-драйвер

Таблица 1.4.2 *(окончание)*

Параметр	**Комментарий**
`/WARN[:`*`warninglevel`*`]`	Определяет количество возможных предупреждений, выдаваемых компоновщиком
`/WS:AGGRESSIVE`	Несколько уменьшает скорость выполнения приложения (Windows NT). Операционная система удаляет данное приложение из памяти в случае его простоя

Программа LINK.EXE и ассемблер могут работать с командными файлами. Например, вместо командной строки `link /windows:console mt.obj` можно создать текстовый файл mtl.cmd со следующим содержанием:

```
/subsystem:console
Mt.obj
```

и выполнить команду `link @mtl.cmd`.

Завершая главу, приведу несколько простых примеров.

Включение в исполняемый файл отладочной информации

Если в исполняемый файл была включена отладочная информация, то фирменный отладчик позволяет работать одновременно и с текстом программы, и с дизассемблированным кодом. Это особенно удобно для языков высокого уровня, а для программ на ассемблере это весьма мощный инструмент отладки.

Пусть текст программы содержится в файле PROG.ASM. Для того чтобы включить отладочную информацию в исполняемый модуль, используем при трансляции для MASM следующие ключи:

```
ML /c /coff / /Zd /Zi prog.asm
LINK /subsystem:windows /debug prog.obj
```

При этом кроме файла PROG.EXE на диске появится файл PROG.PDB, содержащий отладочные данные. Теперь для отладки следует запустить фирменный 32-битный отладчик фирмы Microsoft — Code View или другой отладчик, распознающий отладочную информацию Microsoft.

В качестве примера возьмем программу из листинга 1.4.1. Откомпилируем ее с сохранением (включением) отладочной информации. На рис. 1.4.1 представлено окно отладчика OllyDbg, в который загружена программа с отладочной информацией. Из рисунка видно, что отладчик воспользовался отла-

дочной информацией, в частности именами переменных. Подробнее об отладчике OllyDbg будем говорить в последней части нашей книги.

```
00401074  . 6A 00          PUSH 0                                   hInst = NULL
00401076  . E8 8D010000    CALL 1-18._LoadCursorA@8                 LoadCursorA
0040107B  . A3 34404000    MOV DWORD PTR DS:[404034],EAX
00401080  . C705 38404000  MOV DWORD PTR DS:[404038],11
0040108A  . C705 3C404000  MOV DWORD PTR DS:[40403C],0
00401094  . C705 40404000  MOV DWORD PTR DS:[404040],OFFSET 1-18.Cl  ASCII "CLASS32"
0040109E  . 68 1C404000    PUSH OFFSET 1-18.WC                      pWndClass = OFFSET 1-18.W
004010A3  . E8 72010000    CALL 1-18._RegisterClassA@4              RegisterClassA
004010A8  . 6A 00          PUSH 0                                   lParam = NULL
004010AA  . FF35 44404000  PUSH DWORD PTR DS:[HINST]                hInst = NULL
004010B0  . 6A 00          PUSH 0                                   hMenu = NULL
004010B2  . 6A 00          PUSH 0                                   hParent = NULL
004010B4  . 68 90010000    PUSH 190                                 Height = 190 (400.)
004010B9  . 68 90010000    PUSH 190                                 Width = 190 (400.)
004010BE  . 6A 64          PUSH 64                                  Y = 64 (100.)
004010C0  . 6A 64          PUSH 64                                  X = 64 (100.)
004010C2  . 68 0000CF00    PUSH 0CF0000                             Style = WS_OVERLAPPED|WS_
004010C7  . 68 48404000    PUSH OFFSET 1-18.TITLENAME               WindowName = "Простой при
004010CC  . 68 6D404000    PUSH OFFSET 1-18.CLASSNAME               Class = "CLASS32"
004010D1  . 6A 00          PUSH 0                                   ExtStyle = 0
004010D3  . E8 18010000    CALL 1-18._CreateWindowExA@48            CreateWindowExA
004010D8  . 83F8 00        CMP EAX,0
004010DB  .v74 53          JE SHORT 1-18.00401130
004010DD  . A3 00404000    MOV DWORD PTR DS:[NEWHWND],EAX
004010E2  . 6A 01          PUSH 1                                   ShowState = SW_SHOWNORMAL
004010E4  . FF35 00404000  PUSH DWORD PTR DS:[NEWHWND]              hWnd = NULL
004010EA  . E8 31010000    CALL 1-18._ShowWindow@8                  ShowWindow
004010EF  . FF35 00404000  PUSH DWORD PTR DS:[NEWHWND]              hWnd = NULL
```

Рис. 1.4.1. Окно отладчика OllyDbg

Листинг 1.4.1. Пример простой оконной программы, предназначенной для демонстрации возможностей использования отладочной информации

```
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
;сообщение приходит при закрытии окна
WM_CLOSE            equ 10h
;сообщение приходит при закрытии окна
WM_DESTROY          equ 2
;сообщение приходит при создании окна
WM_CREATE           equ 1
;сообщение при щелчке левой кнопкой мыши в области окна
WM_LBUTTONDOWN      equ 201h
;сообщение при щелчке правой кнопкой мыши в области окна
WM_RBUTTONDOWN      equ 204h
;свойства окна
CS_VREDRAW          equ 1h
CS_HREDRAW          equ 2h
CS_GLOBALCLASS      equ 4000h
```

```
WS_OVERLAPPEDWINDOW  equ 000CF0000H
style equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
;идентификатор стандартной пиктограммы
IDI_APPLICATION      equ 32512
;идентификатор курсора
IDC_CROSS            equ 32515
;режим показа окна - нормальный
SW_SHOWNORMAL        equ 1
;прототипы внешних процедур
EXTERN  MessageBoxA@16:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;структуры
;структура сообщения
MSGSTRUCT STRUC
MSHWND         DD ?   ; идентификатор окна, получающего сообщение
MSMESSAGE      DD ?   ; идентификатор сообщения
MSWPARAM       DD ?   ; дополнительная информация о сообщении
MSLPARAM       DD ?   ; дополнительная информация о сообщении
MSTIME         DD ?   ; время посылки сообщения
MSPT           DD ?   ; положение курсора во время посылки сообщения
MSGSTRUCT ENDS
;---------
WNDCLASS STRUC
CLSSTYLE       DD ?   ; стиль окна
CLWNDPROC      DD ?   ; указатель на процедуру окна
CLSCEXTRA      DD ?   ; информация о дополнительных байтах
                      ; для данной структуры
CLWNDEXTRA     DD ?   ; информация о дополнительных байтах для окна
CLSHINSTANCE   DD ?   ; дескриптор приложения
CLSHICON       DD ?   ; идентификатор пиктограммы окна
```

```
CLSHCURSOR    DD ?   ; идентификатор курсора окна
CLBKGROUND    DD ?   ; идентификатор кисти окна
CLMENUNAME    DD ?   ; имя-идентификатор меню
CLNAME        DD ?   ; специфицирует имя класса окон
WNDCLASS ENDS
;сегмент данных
_DATA SEGMENT
NEWHWND       DD 0
MSG           MSGSTRUCT <?>
WC            WNDCLASS  <?>
HINST         DD 0 ;здесь хранится дескриптор приложения
TITLENAME     DB 'Простой пример 32-битного приложения',0
CLASSNAME     DB 'CLASS32',0
CAP           DB 'Сообщение',0
MES1          DB 'Выход из программы. Пока!',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH    0
      CALL    GetModuleHandleA@4
      MOV     [HINST], EAX
REG_CLASS:
;заполнить структуру окна
; стиль
      MOV    [WC.CLSSTYLE],style
;процедура обработки сообщений
      MOV     WC.CLWNDPROC, OFFSET WNDPROC
      MOV     WC.CLSCEXTRA, 0
      MOV     WC.CLWNDEXTRA, 0
      MOV     EAX, [HINST]
      MOV     WC.CLSHINSTANCE, EAX
;----------пиктограмма окна
      PUSH    IDI_APPLICATION
      PUSH    0
      CALL    LoadIconA@8
      MOV     WC.CLSHICON, EAX
;----------курсор окна
      PUSH    IDC_CROSS
      PUSH    0
      CALL    LoadCursorA@8
      MOV     WC.CLSHCURSOR, EAX
;----------
      MOV     WC.CLBKGROUND, 17 ;цвет окна
      MOV     DWORD PTR WC.CLMENUNAME,0
```

```
      MOV     DWORD PTR WC.CLNAME, OFFSET CLASSNAME
      PUSH    OFFSET WC
      CALL    RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH    0
      PUSH    [HINST]
      PUSH    0
      PUSH    0
      PUSH    400    ; DY - высота окна
      PUSH    400    ; DX - ширина окна
      PUSH    100    ; Y-координата левого верхнего угла
      PUSH    100    ; X-координата левого верхнего угла
      PUSH    WS_OVERLAPPEDWINDOW
      PUSH    OFFSET TITLENAME   ; имя окна
      PUSH    OFFSET CLASSNAME   ; имя класса
      PUSH    0
      CALL    CreateWindowExA@48
;проверка на ошибку
      CMP     EAX,0
      JZ     _ERR
      MOV     NEWHWND, EAX    ; дескриптор окна
;-----------------------------------
      PUSH    SW_SHOWNORMAL
      PUSH    NEWHWND
      CALL    ShowWindow@8    ; показать созданное окно
;-----------------------------------
      PUSH    NEWHWND
      CALL    UpdateWindow@4  ; команда перерисовать видимую
                              ; часть окна, сообщение WM_PAINT
;цикл обработки сообщений
MSG_LOOP:
      PUSH    0
      PUSH    0
      PUSH    0
      PUSH    OFFSET MSG
      CALL    GetMessageA@16
      CMP     EAX, 0
      JE      END_LOOP
      PUSH    OFFSET MSG
      CALL    TranslateMessage@4
      PUSH    OFFSET MSG
      CALL    DispatchMessageA@4
      JMP     MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
      PUSH    MSG.MSWPARAM
```

```
      CALL    ExitProcess@4
_ERR:
      JMP     END_LOOP
;-------------------------------------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H] LPARAM
; [EBP+10H] WAPARAM
; [EBP+0CH] MES
; [EBP+8] HWND
WNDPROC PROC
      PUSH    EBP
      MOV     EBP, ESP
      PUSH    EBX
      PUSH    ESI
      PUSH    EDI
      CMP     DWORD PTR [EBP+0CH], WM_DESTROY
      JE      WMDESTROY
      CMP     DWORD PTR [EBP+0CH],WM_CLOSE ;закрытие окна
      JE      WMCLOSE
      CMP     DWORD PTR [EBP+0CH], WM_CREATE
      JE      WMCREATE
      JMP     DEFWNDPROC
;нажатие правой кнопки мыши приводит к закрытию окна
RBUTTON:
      JMP     WMDESTROY
WMCREATE:
      MOV     EAX, 0
      JMP     FINISH
WMCLOSE:
DEFWNDPROC:
      PUSH    DWORD PTR [EBP+14H]
      PUSH    DWORD PTR [EBP+10H]
      PUSH    DWORD PTR [EBP+0CH]
      PUSH    DWORD PTR [EBP+08H]
      CALL    DefWindowProcA@16
      JMP     FINISH
WMDESTROY:
      PUSH    0          ; MB_OK
      PUSH    OFFSET CAP
      PUSH    OFFSET MES1
      PUSH    DWORD PTR [EBP+08H]   ; дескриптор окна
      CALL    MessageBoxA@16
      PUSH    0
      CALL    PostQuitMessage@4     ; сообщение WM_QUIT
      MOV     EAX, 0
```

```
FINISH:
      POP     EDI
      POP     ESI
      POP     EBX
      POP     EBP
      RET     16
WNDPROC ENDP
_TEXT ENDS
END START
```

Трансляция программы:

```
ML /c /coff / /Zd /Zi prog.asm
LINK /subsystem:windows /debug prog.obj
```

Получение консольных и GUI-приложений

О консольных приложениях речь еще впереди, здесь же я буду краток. Консольные приложения — это приложения, работающие с текстовым экраном, при этом они являются полноценными 32-битными приложениями. О структуре консольных программ речь пойдет позже, сейчас же замечу, что для получения консольного приложения с помощью LINK.EXE следует использовать ключ `/subsystem:console` вместо `/subsystem:windows`.

Автоматическая компоновка

Транслятор ML.EXE обладает удобным свойством автоматического запуска компоновщика. Обычно мы игнорируем это свойство, используя ключ `/c`. Если не применять это ключ, то транслятор ML будет пытаться запустить программу LINK.EXE. Чтобы правильно провести всю трансляцию, необходимо еще указать опции компоновщика. Вот как будет выглядеть вся строка:

```
ML /coff prog.asm /LINK /subsystem:windows
```

Не правда ли, удобно?

"Самотранслирующаяся" программа

Существует один очень интересный прием, позволяющий оформлять текст программы таким образом, чтобы ее можно было бы запускать как пакетный (batch) файл, который транслировал бы сам себя. Этот прием, собственно, основывается на том факте, что командный процессор игнорирует такой

знак, как точку с запятой. А поскольку для ассемблера этой знак используется для обозначения начала строкового комментария, то мы всегда можем скрывать команды операционной системы от ассемблера. В листинге 1.4.2 приведен скелет такой программы, и не забудьте, что она должна иметь расширение bat. В нашем случае назовем ее M.BAT.

Листинг 1.4.2. Скелет самотранслирующейся программы

```
;rem пример самотранслирующейся программы
;goto masm
;здесь текст программы
.
.
.
END START

;:masm
;ml /c /coff M.BAT
;link /subsysytem:windows M.OBJ
```

Вот, собственно, и все. Как видите — мелочь, но из таких "жемчужинок" и состоит искусство программирования.

Глава 1.5

О кодировании текстовой информации в операционной системе Windows

Кодировка — дело важное. Еще в доисторические времена, когда о Windows ходили лишь смутные слухи, программистам приходилось заниматься перекодированием текстовой информации. Часто необходимо было писать драйвер для принтера, который не хотел понимать обычную кодировку. Иногда возникали проблемы с чтением текстов, написанных в различных кодировках. В этом случае использовали многочисленные программы-перекодировщики, которые в то время писал чуть ли не каждый второй программист. Но времена меняются, и, кажется, в этом вопросе мы приходим к более ясному и стабильному состоянию.

О кодировании текстовой информации

Институтом стандартизации США (American National Standard Institute, ANSI) была введена система кодирования текстовой информации ASCII (American Standard Code for Information Interchange). В системе ASCII существуют две таблицы кодирования — базовая и расширенная. В базовую таблицу входят значения кодов от 0 до 127, а в расширенную — значения от 128 до 255. Первые 32 кода базовой таблицы отведены для использования производителями аппаратных средств и представляют собой так называемые управляющие коды. Коды 32—127 применяются для представления символов английского алфавита, цифр, знаков препинания и других символов.

Расширенная таблица отводилась для кодирования национальных алфавитов. Здесь, по сути, отсутствует какой-либо стандарт[1]. На территории России дей-

[1] В стандарте ISO (International Standard Organization) предусмотрено кодирование букв русского алфавита, но используется этот стандарт крайне редко.

ствует несколько таких кодировок. Так, например, для кодирования символов русского языка фирмой Microsoft была введена кодировка Windows 1251, кодировка КОИ-8 используется для кодирования русских букв, еще со времен Советского Союза, кодировка DOS или CP-866 обычно применяется в Windows для отображения текстовой информации в окне консоли[2].

Однобайтовое кодирование не позволяет охватывать более двух различных алфавитов одновременно. Более того, некоторые алфавиты невозможно закодировать при помощи однобайтовых чисел. В настоящее время все чаще используется универсальное кодирование, основанное на представление знаков с помощью двухбайтовых чисел. Такая система называется универсальной или Unicode. Заметим, что, несмотря на очевидную выгоду такого подхода, использование двухбайтового кодирования получило широкое распространение сравнительно недавно. Причина тому очень простая — для использования кодировки Unicode требуются дополнительные ресурсы: это касается как памяти — все текстовые строки удваиваются, так и производительности процессора.

OEM и ANSI

В Windows применяются два типа кодировок:

- кодировка, используемая для вывода текстовой информации в графических окнах. Ее называют еще ANSI-кодировкой;
- кодировка, используемая для вывода информации в консольные окна. Ее также называют OEM-кодировкой (Original Equipment Manufacture).

Лишний раз подчеркну, что в качестве ANSI или OEM в принципе может выступать любая кодировка. Важно, что в Windows для графических и консольных окон используются разные кодировки. При этом, как было сказано ранее, раскладка кодов от 0 до 127 у разных кодировок совпадает. Следовательно, проблема может возникнуть, если информация, которую мы собрались выводить в окно, будет содержать текст на русском языке. В Windows имеются средства, позволяющие преобразовывать строки из одного вида кодировки в другой. Чаще всего используют API-функции `OemToChar` и `CharToOem`. Рассмотрим эти функции подробнее, т. к. впоследствии нам неоднократно придется к ним обращаться. В Си-нотации эти функции выглядят следующим образом.

```
BOOL OemToChar
(
```

[2] Одной из задач консольного окна и было отображение работы DOS-программ.

```
LPCSTR lpszSrc,
LPTSTR lpszDst
)

BOOL CharToOem
(
LPCTSTR lpszSrc,
LPSTR lpszDst
)
```

Функция `OemToChar` преобразует строку из кодировки, используемой для консольного вывода, в строку в кодировке, используемой для вывода в графическое окно. Функция `CharToOem` осуществляет обратное преобразование. Первым аргументом обеих функций является адрес буфера, куда будет производиться копирование перекодированной строки. Вторым аргументом является адрес строки, которая будет подвергнута перекодированию. На практике удобно совмещать оба буфера, т. е. переводить строку из одной кодировки в другую. Для этого и первый, и второй параметры функций должны совпадать, т. е. указывать на одну и ту же строку. Замечу, что буфер-источник должен обязательно заканчиваться символом, имеющим код 0.

Кодировка Unicode

Операционные системы семейства Windows NT, начиная с Windows 2000, полностью переведены на кодировку Unicode. Для многих программистов это прошло, однако, полностью незамеченным. Дело в том, что с Unicode работают внутренние процедуры Windows. Входные же параметры, например строки для функции `MessageBox`, ОС по-прежнему воспринимает в кодировке ANSI. При вызове такой функции операционная система преобразует входную ANSI-строку к двухбайтовому виду и затем работает с такой строкой. Если функция должна возвращать строку, то строка должна дополнительно быть преобразована из Unicode в ANSI. Кроме этого, для функций, получающих или возвращающих строки, имеются "двойники" с тем же именем, в конце которого добавлена буква "W", например, `MessageBoxW`, `CharToOemW` и т. п. Эти функции изначально оперируют строками в кодировке Unicode, не перекодируя их во внутренний формат и обратно. Ресурсы, о которых речь пойдет в дальнейшем, также хранятся в кодировке Unicode. Следовательно, все функции, помещающие и извлекающие текстовую информацию из ресурсов, осуществляют предварительное перекодирование.

Очень интересна функция `IsTextUnicode`, которая определяет, является информация в данном буфере в кодировке Unicode или нет. Функция статисти-

ческая, т. е. определяет, является данный текст в кодировке Unicode или нет с некоторой вероятностью. Рассмотрим функцию подробнее.

```
BOOL IsTextUnicode
(
CONST VOID* pBuffer,
int cb,
LPINT lpi
)
```

Функция возвращает ненулевое значение, если тест завершился успешно в пользу кодировки Unicode, и ноль в противном случае.

Рассмотрим параметры функции:

- 1-й параметр является адресом буфера, который содержит текстовую информацию и будет подвергнут испытанию;
- 2-й параметр содержит длину тестируемого буфера;
- 3-й параметр — это указатель на область памяти, куда надо поместить указание, какие тесты следует провести. Если в области памяти содержится 0, то это означает, что следует провести все тесты. Например, значение `IS_TEXT_UNICODE_ASCII16`, равное 1, подразумевает, что представленный текст должен быть в кодировке Unicode и содержать символы из национального алфавита (например, русского). Другие значения констант можно найти, например, в файле WINDOWS.INC, прилагаемом с пакетом MASM32. Существенно то, что все константы содержат неперекрывающиеся биты, т. е. в область памяти, на которую указывает третий аргумент, можно помещать комбинации этих констант. Пример использования данной функции я приведу далее.

Рассмотрим теперь, как производится преобразование строки из ANSI-кодировки в Unicode и обратно. Для этого используются две функции: `MultiByteToWideChar` и `WideCharToMultiByte`. Рассмотрим их более подробно.

Функция `MultiByteToWideChar` служит для преобразования строки в кодировке ANSI в строку в кодировке Unicode.

```
int MultiByteToWideChar(
UINT CodePage,
DWORD dwFlags,
LPCSTR lpMultiByteStr,
int cbMultiByte,
LPWSTR lpWideCharStr,
int cchWideChar
)
```

Параметры функции:

- 1-й параметр задает номер кодовой страницы, относящийся к исходной строке. Например, константа `CP_ACP = 0` означает ASCII-кодировку;
- 2-й параметр — флаг, который влияет на преобразование букв с диакритическими знаками. Данный параметр обычно полагают равным 0;
- 3-й параметр указывает на преобразуемую строку;
- 4-й параметр должен быть равен длине преобразуемой строки. Если положить данный параметр –1, то функция должна сама определить длину преобразуемой строки;
- 5-й параметр указывает на буфер, куда будет помещена преобразованная строка;
- 6-й параметр задает максимальный размер буфера, куда будет помещена преобразованная строка.

При успешном завершении функция возвращает размер преобразованной строки. Если шестой параметр положить равным 0, то функция не будет производить преобразование, а возвратит размер буфера, который необходим для преобразованной строки.

```
int WideCharToMultiByte(
  UINT CodePage,
  DWORD dwFlags,
  LPCWSTR lpWideCharStr,
  int cchWideChar,
  LPSTR lpMultiByteStr,
  int cbMultiByte,
  LPCSTR lpDefaultChar,
  LPBOOL lpUsedDefaultChar
)
```

Параметры функции:

- 1-й параметр определяет кодовую страницу для результирующей строки;
- 2-й параметр — флаг, который влияет на преобразование букв с диакритическими знаками. Данный параметр обычно полагают равным нулю;
- 3-й параметр — это адрес преобразуемой строки;
- 4-й параметр — длина в символах преобразуемой строки. Если параметр задать равным –1, то функция самостоятельно будет определять длину преобразуемой строки;
- 5-й параметр — адрес буфера, куда будет помещена результирующая ANSI-строка;

- 6-й параметр определяет максимальный размер буфера для преобразованной строки;
- 7-й параметр — адрес символа по умолчанию. Если функция встречает символ, отсутствующий в указанной кодовой странице, то она берет символ, на который указывает данный параметр. Обычно, однако, его полагают равным 0 (т. е. `NULL`);
- 8-й параметр, указывает на область памяти, куда функция помещает 0 или 1, в зависимости от того, удалось или нет преобразовать все символы в исходной строке.

В листинге 1.5.1 показан фрагмент программы, демонстрирующий преобразование строки из кодировки ANSI в кодировку Unicode.

Листинг 1.5.1. Фрагмент, преобразующий строку в кодировке ANSI в кодировку Unicode

```
.
.
;преобразуемая строка
STR1 DB "Консольное приложение",0
;буфер, для копирования преобразованной строки
;в кодировке Unicode
BUF  DB 200 DUP(0)
.
.
.
PUSH 200               ;максимальная длина буфера
PUSH OFFSET BUF        ;адрес буфера
PUSH -1                ;определять длину автоматически
PUSH OFFSET STR1       ;адрес строки
PUSH 0                 ;флаг
PUSH 0                 ;CP_ACP - кодировка ANSI
CALL MultiByteToWideChar@24
;далее можно работать со строкой в кодировке Unicode — BUF
...
```

По поводу представленного выше фрагмента замечу, что более корректно следовало бы действовать так:

1. Запустить функцию `MultiByteToWideChar`, положив значение 6-го параметра равным 0. Функция при этом возвратит размер буфера в байтах для преобразованной строки.
2. Выделить объем памяти для указанного буфера.

3. Осуществить преобразование строки, вызвав функцию `MultiByteToWideChar` с указанием значения 6-го параметра.
4. Поработать со строкой.
5. Освободить выделенный под буфер блок памяти.

Я надеюсь, что читатель, познакомившись в *главе 3.6* с основами управления памятью Windows, вернется к вопросу перекодировки и, воспользовавшись представленным выше алгоритмом, напишет свою программу перекодировки.

В *главе 2.7* можно найти интересный макрос, упрощающий преобразование строки из кодировки ASCII в кодировку Unicode.

Наконец, укажу один весьма полезный прием задания прямо в программе строки, которая будет восприниматься как раз в кодировке Unicode. К примеру, вместо задания строки традиционным способом, т. е. скажем так:

```
STR1 DB "MASM 9.0",0
```

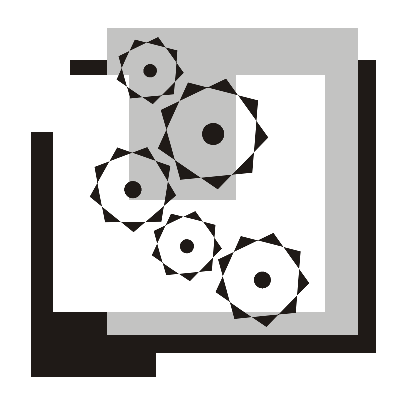
можно записать так:

```
STR1 DW 'M', 'A', 'S', 'M', '9', '.', '0',0
```

и спокойно использовать функцию `MessageBoxW` вместо функции `MessageBoxA`.

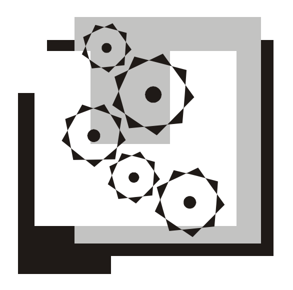
Замечание

Среди функций API, относящихся к кодированию текстовой информации, я бы выделил еще две функции: `CharToOemBuff` и `OemToCharBuff`. Эти функции имеют две особенности. Во-первых, у них третий параметр показывает, сколько символов (именно символов, а не байтов) из буфера-источника следует преобразовать. При этом в число преобразуемых символов могут войти и символы с кодом 0 (кодом конца строки). Во-вторых, обе функции имеют две разновидности: с постфиксом `A` (ANSI) и с постфиксом `W` (Wide Char). Таким образом, например, функция `CharToOemBuffA` преобразует строку из кодировки ANSI в кодировку OEM, а функция `CharToOemBuffW` преобразует строку в кодировке Unicode в кодировку OEM.



Часть II

ПРОСТЫЕ ПРОГРАММЫ, КОНСОЛЬНЫЕ ПРИЛОЖЕНИЯ, ОБРАБОТКА ФАЙЛОВ



Глава 2.1

Вывод графики и текста в окно. Библиотека GDI

Примеры — это фундамент обучения программированию. На примерах учился и я. В данной главе мы серьезно начинаем работать с сообщением `WM_PAINT`. В *главе 1.3* мы уже рассматривали это сообщение, но не использовали его в своих программах. Причиной являлось то, что в окне у нас были лишь элементы управления, но не было текстовой информации и графических изображений. Теперь мы исправим положение. Кроме обработки сообщения `WM_PAINT`, речь в этой главе пойдет и о других проблемах, возникающих при программировании в Windows.

Вывод текста в окне

Основным средством для вывода графической информации в Windows является GDI (Graphics Device Interface, интерфейс графического устройства). С GDI тесно связано такое понятие, как контекст устройства. *Контекст устройства* (Device Context, DC) представляет собой некоторую структуру Windows, предназначенную для вывода текстовой или графической информации. Если вы хотите рисовать в окне или на экране, то вам не обойтись без контекста окна или экрана. Получив контекст, точнее, его дескриптор (число), вы можете использовать его для вывода текстовой и графической информации. Для подключения API-функций GDI нам будет необходимо подключить библиотеку gdi32.lib (см. листинг 2.1.1).

Если вы только начинаете программировать под Windows, то в программе из листинга 2.1.1 найдете много нового. Поэтому приступим к подробному разъяснению этой программы.

- В данной программе мы определяем цвет окна и текста через комбинацию значений трех цветов: красного, зеленого и синего. Цвет определяется

одним 32-битным числом. В этом числе первый байт — интенсивность красного, второй байт — интенсивность зеленого, третий байт — интенсивность синего цвета. Последний байт равен нулю. Механизм получения этого числа продемонстрирован в определении константы `RGBW`.

- Цвет окна задается посредством определения кисти через функцию `CreateSolidBrush`. Кисть — это битовый шаблон, который используется системой для заполнения сплошных областей.
- Поскольку при перерисовке окна системой посылается сообщение `WM_PAINT`, именно при получении этого сообщения и следует перерисовывать содержимое окна. В данном случае мы выводим всего лишь одну строку текста. Для того чтобы осуществить вывод информации в окно, необходимо сначала получить контекст окна (контекст устройства — Device Context). Для нас это просто некоторое число (дескриптор контекста устройства), посредством которого осуществляется связь между приложением и окном. Обычно контекст устройства определяется посредством функции `GetDC`. При получении сообщения `WM_PAINT` контекст устройства следует получать посредством функции `BeginPaint`. Аргументом для нее является указатель на специальную структуру, которая у нас называется `PAINTSTR` и поля которой, впрочем, мы пока не используем.
- Текст выводится посредством функции `OutText`. Предварительно, с помощью функций `SetBkColor` и `SetTextColor`, мы определяем цвет фона и цвет букв. Цвет фона в нашей программе, соответственно, совпадает с цветом окна. Поскольку текст выводится при получении сообщения `WM_PAINT`, он не теряется при перекрытии окна другими окнами, при свертывании/развертывании окна.
- Несколько слов о системе координат. Центр системы координат находится в левом верхнем углу, ось y направлена вниз, ось x — вправо. Впрочем, это общепринятый вариант для графических экранов.
- Еще один момент также связан с выводом текста в окно. Одним из параметров функции `OutText` является количество символов выводимой строки. И вот здесь начинается интересное. Определить длину строки (за минусом нулевого элемента) можно по-разному. Например, можно использовать операторы макроассемблера `SIZEOF`, `LENGTHOF`, `LENGTH` или `SIZE`. Но раз уже мы употребляем определение строк, как это принято в Си, — естественно и определить функции для работы со строковыми переменными (см. замечание в конце главы) такого типа. В данном примере мы определили функцию `LENSTR`, которая возвращает длину строки. Не смущайтесь, что функция помещает результат в регистр `EBX`. Нам просто так удобнее. У функции, кроме того, есть одно очень важное преимущество перед мак-

росредствами — она получает длину при выполнении программы, а не во время ее трансляции. Функция `LENSTR` получает в качестве параметра адрес строки. Параметр передается в функцию через стек (см. в листинге 2.1.1 команду `PUSH OFFSET TEXT` перед вызовом функции `LENSTR`).

Листинг 2.1.1. Пример простейшей программы вывода текста в окно

```
;файл text1.inc
;константы
;сообщение приходит при закрытии окна
WM_DESTROY      equ 2
;сообщение приходит при создании окна
WM_CREATE       equ 1
;сообщение приходит при перерисовке окна
WM_PAINT        equ 0Fh
;свойства окна
CS_VREDRAW      equ 1h
CS_HREDRAW      equ 2h
CS_GLOBALCLASS  equ 4000h
WS_OVERLAPPEDWINDOW equ 000CF0000H
stylcl          equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
DX0             equ 300
DY0             equ 200
;компоненты цветов
RED             equ 50
GREEN           equ 50
BLUE            equ 255
RGBW            equ (RED or (GREEN shl 8)) or (BLUE shl 16)
RGBT            equ 255 ;красный
;идентификатор стандартной пиктограммы
IDI_APPLICATION equ 32512
;идентификатор курсора
IDC_CROSS       equ 32515
;режим показа окна - нормальный
SW_SHOWNORMAL   equ 1
;прототипы внешних процедур
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
```

```
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
EXTERN  BeginPaint@8:NEAR
EXTERN  EndPaint@8:NEAR
EXTERN  TextOutA@20:NEAR
EXTERN  GetStockObject@4:NEAR
EXTERN  CreateSolidBrush@4:NEAR
EXTERN  SetBkColor@8:NEAR
EXTERN  SetTextColor@8:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND      DD ?   ; идентификатор окна, получающего сообщение
      MSMESSAGE   DD ?   ; идентификатор сообщения
      MSWPARAM    DD ?   ; дополнительная информация о сообщении
      MSLPARAM    DD ?   ; дополнительная информация о сообщении
      MSTIME      DD ?   ; время посылки сообщения
      MSPT        DD ?   ; положение курсора во время посылки сообщения
MSGSTRUCT   ENDS
;---------
WNDCLASS STRUC
      CLSSTYLE       DD ? ;стиль окна
      CLSLPFNWNDPROC DD ? ;указатель на процедуру окна
      CLSCBCLSEXTRA  DD ? ;информация о дополнительных байтах
                          ;для данной структуры
      CLSCBWNDEXTRA  DD ? ;информация о дополнительных байтах для окна
      CLSHINSTANCE   DD ? ;дескриптор приложения
      CLSHICON       DD ? ;идентификатор пиктограммы окна
      CLSHCURSOR     DD ? ;идентификатор курсора окна
      CLSHBRBACKGROUND  DD ?;идентификатор кисти окна
      MENNAME        DD ? ;имя-идентификатор меню
      CLSNAME        DD ? ;специфицирует имя класса окон
WNDCLASS  ENDS
;---
PAINTSTR STRUC
      hdc     DWORD 0
      fErase  DWORD 0
      left    DWORD 0
      top     DWORD 0
      right   DWORD 0
      bottom  DWORD 0
      fRes    DWORD 0
      fIncUp  DWORD 0
      Reserv  DB 32 dup(0)
```

```
PAINTSTR  ENDS
;файл text1.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;-------------------------------------------------
;подключение внешнего текста
include text1.inc
;подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\gdi32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      NEWHWND    DD 0
      MSG        MSGSTRUCT <?>
      WC         WNDCLASS  <?>
      PNT        PAINTSTR  <?>
      HINST      DD 0
      TITLENAME  DB 'Текст в окне',0
      NAM        DB 'CLASS32',0
      XT         DWORD 30
      YT         DWORD 30
      TEXT       DB 'Текст в окне красный',0
_DATA  ENDS
;сегмент кода
_TEXT  SEGMENT
START:
;получить дескриптор приложения
      PUSH    0
      CALL    GetModuleHandleA@4
      MOV     HINST, EAX
REG_CLASS:
;заполнить структуру окна
;стиль
      MOV     WC.CLSSTYLE,stylcl
;процедура обработки сообщений
      MOV     WC.CLSLPFNWNDPROC,OFFSET WNDPROC
      MOV     WC.CLSCBCLSEXTRA,0
      MOV     WC.CLSCBWNDEXTRA,0
      MOV     EAX,HINST
      MOV     WC.CLSHINSTANCE,EAX
; ----------пиктограмма окна
      PUSH    IDI_APPLICATION
      PUSH    0
```

```
      CALL    LoadIconA@8
      MOV     WC.CLSHICON, EAX
;----------курсор окна
      PUSH    IDC_CROSS
      PUSH    0
      CALL    LoadCursorA@8
      MOV     WC.CLSHCURSOR, EAX
;----------
      PUSH    RGBW                 ; цвет кисти
      CALL    CreateSolidBrush@4   ; создать кисть
      MOV     WC.CLSHBRBACKGROUND,EAX
      MOV     DWORD PTR WC.MENNAME,0
      MOV     DWORD PTR WC.CLSNAME,OFFSET NAM
      PUSH    OFFSET WC
      CALL    RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH    0
      PUSH    HINST
      PUSH    0
      PUSH    0
      PUSH    DY0     ; DY0 - высота окна
      PUSH    DX0     ; DX0 - ширина окна
      PUSH    100     ; координата Y
      PUSH    100     ; координата X
      PUSH    WS_OVERLAPPEDWINDOW
      PUSH    OFFSET TITLENAME   ; имя окна
      PUSH    OFFSET NAM         ; имя класса
      PUSH    0
      CALL    CreateWindowExA@48
;проверка на ошибку
      CMP     EAX,0
      JZ      _ERR
      MOV     NEWHWND, EAX   ; дескриптор окна
;-----------------------------------
      PUSH    SW_SHOWNORMAL
      PUSH    NEWHWND
      CALL    ShowWindow@8   ; показать созданное окно
;-----------------------------------
      PUSH    NEWHWND
      CALL    UpdateWindow@4 ; перерисовать видимую часть окна
;цикл обработки сообщений
MSG_LOOP:
      PUSH    0
      PUSH    0
      PUSH    0
      PUSH    OFFSET MSG
```

```
      CALL    GetMessageA@16
      CMP     AX, 0
      JE      END_LOOP
      PUSH    OFFSET MSG
      CALL    TranslateMessage@4
      PUSH    OFFSET MSG
      CALL    DispatchMessageA@4
      JMP     MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
      PUSH    [MSG.MSWPARAM]
      CALL    ExitProcess@4
_ERR:
      JMP     END_LOOP
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC   PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    EBX
      PUSH    ESI
      PUSH    EDI
      CMP     DWORD PTR [EBP+0CH],WM_DESTROY
      JE      WMDESTROY
      CMP     DWORD PTR [EBP+0CH],WM_CREATE
      JE      WMCREATE
      CMP     DWORD PTR [EBP+0CH],WM_PAINT
      JE      WMPAINT
      JMP     DEFWNDPROC
;обработка сообщения WM_PAINT
WMPAINT:
;----------------
      PUSH    OFFSET PNT
      PUSH    DWORD PTR [EBP+08H]
      CALL    BeginPaint@8
      PUSH    EAX ;сохранить контекст (дескриптор)
;---------------- цвет фона = цвет окна
      PUSH    RGBW
      PUSH    EAX
      CALL    SetBkColor@8
;---------------- контекст
      POP     EAX
```

```
      PUSH    EAX
;---------------- цвет текста (красный)
      PUSH    RGBT
      PUSH    EAX
      CALL    SetTextColor@8
;---------------- дескриптор контекста в регистр EAX
      POP     EAX
;---------------- вывести текст
;вначале получим длину строки
      PUSH    OFFSET TEXT
      CALL    LENSTR
      PUSH    EBX         ; длина строки
      PUSH    OFFSET TEXT ; адрес строки
      PUSH    YT          ; Y
      PUSH    XT          ; X
      PUSH    EAX         ; дескриптор контекста окна
      CALL    TextOutA@20
;---------------- закрыть
      PUSH    OFFSET PNT
      PUSH    DWORD PTR [EBP+08H]
      CALL    EndPaint@8
      MOV     EAX, 0
      JMP     FINISH
WMCREATE:
      MOV     EAX, 0
      JMP     FINISH
DEFWNDPROC:
      PUSH    DWORD PTR [EBP+14H]
      PUSH    DWORD PTR [EBP+10H]
      PUSH    DWORD PTR [EBP+0CH]
      PUSH    DWORD PTR [EBP+08H]
      CALL    DefWindowProcA@16
      JMP     FINISH
WMDESTROY:
      PUSH    0
      CALL    PostQuitMessage@4 ;WM_QUIT
      MOV     EAX, 0
FINISH:
      POP     EDI
      POP     ESI
      POP     EBX
      POP     EBP
      RET     16
WNDPROC ENDP
;----------- функция получения длины строки ------------
;[EBP+08H] - указатель на строку
```

```
;в EBX возвращается длина строки
LENSTR  PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    ESI
      MOV     ESI,DWORD PTR [EBP+8]
      XOR     EBX,EBX
LBL1:
;проверка, не конец ли строки
      CMP     BYTE PTR [ESI],0
      JZ      LBL2
      INC     EBX
      INC     ESI
      JMP     LBL1
LBL2:
      POP     ESI
      POP     EBP
      RET     4
LENSTR  ENDP
_TEXT  ENDS
END START
```

Трансляция программы из листинга 2.1.1:

```
ml /c /coff  text1.asm
link /subsystem:windows text1.obj
```

Рассмотрим еще один пример с выводом текста в окно (листинг 2.1.2). Теперь мы усложняем задачу. Зададимся целью, чтобы текстовая строка все время, что бы ни происходило с окном, была бы в его середине. Для этого необходимо знать длину строки в пикселах (экранных точках) и размеры окна. Длина строки в пикселах определяется с помощью функции `GetTextExtentPoint32`, а размеры окна — с помощью функции `GetWindowRect`. При этом нам понадобятся структуры типа `SIZET` и `RECT`. Надеюсь, читатель понимает, как определить положение строки, если известна ее длина и размеры окна. Добавлю только, что еще необходимо учесть высоту заголовка окна.

Листинг 2.1.2. Текстовая строка все время в середине окна

```
;файл text2.inc
;константы
;сообщение приходит при закрытии окна
WM_DESTROY     equ 2
;сообщение приходит при создании окна
WM_CREATE      equ 1
```

```
;сообщение приходит при перерисовке окна
WM_PAINT      equ 0Fh
;свойства окна
CS_VREDRAW    equ 1h
CS_HREDRAW    equ 2h
CS_GLOBALCLASS      equ 4000h
WS_OVERLAPPEDWINDOW equ 000CF0000H
stylcl        equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
DX0           equ 300
DY0           equ 200
;компоненты цветов
RED           equ 80
GREEN         equ 80
BLUE          equ 255
RGBW          equ (RED or (GREEN shl 8)) or (BLUE shl 16)
RGBT          equ 00FF00H ;зеленый
;идентификатор стандартной пиктограммы
IDI_APPLICATION  equ 32512
;идентификатор курсора
IDC_CROSS     equ 32515
;режим показа окна - нормальный
SW_SHOWNORMAL equ 1
;прототипы внешних процедур
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
EXTERN  BeginPaint@8:NEAR
EXTERN  EndPaint@8:NEAR
EXTERN  TextOutA@20:NEAR
EXTERN  GetStockObject@4:NEAR
EXTERN  CreateSolidBrush@4:NEAR
EXTERN  SetBkColor@8:NEAR
EXTERN  SetTextColor@8:NEAR
EXTERN  GetTextExtentPoint32A@16:NEAR
EXTERN  GetWindowRect@8:NEAR
;структуры
```

```
;структура сообщения
MSGSTRUCT STRUC
      MSHWND          DD ?   ; идентификатор окна, получающего сообщение
      MSMESSAGE       DD ?   ; идентификатор сообщения
      MSWPARAM        DD ?   ; дополнительная информация о сообщении
      MSLPARAM        DD ?   ; дополнительная информация о сообщении
      MSTIME          DD ?   ; время посылки сообщения
      MSPT            DD ?   ; положение курсора, во время
                             ; посылки сообщения
MSGSTRUCT ENDS
;---------
WNDCLASS STRUC
      CLSSTYLE        DD ?  ; стиль окна
      CLSLPFNWNDPROC DD ?  ; указатель на процедуру окна
      CLSCBCLSEXTRA   DD ?  ; информация о дополнительных байтах
                            ; для данной структуры
      CLSCBWNDEXTRA   DD ?  ; информация о дополнительных байтах для окна
      CLSHINSTANCE    DD ?  ; дескриптор приложения
      CLSHICON        DD ?  ; идентификатор пиктограммы окна
      CLSHCURSOR      DD ?  ; идентификатор курсора окна
      CLSHBRBACKGROUND  DD ?  ; идентификатор кисти окна
      MENNAME         DD ?  ; имя-идентификатор меню
      CLSNAME         DD ?  ; специфицирует имя класса окон
WNDCLASS ENDS
;----------
PAINTSTR STRUC
      hdc     DWORD 0
      fErase  DWORD 0
      left    DWORD 0
      top     DWORD 0
      right   DWORD 0
      bottom  DWORD 0
      fRes    DWORD 0
      fIncUp  DWORD 0
      Reserv  DB 32 dup(0)
PAINTSTR  ENDS
;------------
SIZET STRUC
      X1      DWORD ?
      Y1      DWORD ?
SIZET ENDS
RECT STRUC
      L DWORD ? ;X-координата левого верхнего угла
      T DWORD ? ;Y-координата левого верхнего угла
      R DWORD ? ;X-координата правого нижнего угла
      B DWORD ? ;Y-координата правого нижнего угла
```

```
RECT  ENDS
;файл text2.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;-----------------------------------------------
include text2.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\gdi32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      NEWHWND DD 0
      MSG     MSGSTRUCT <?>
      WC      WNDCLASS  <?>
      PNT     PAINTSTR  <?>
      SZT     SIZET   <?>
      RCT     RECT    <?>
      HINST   DD 0
      TITLENAME DB 'Текст в окне',0
      NAM     DB 'CLASS32',0
      XT      DWORD ?
      YT      DWORD ?
      TEXT    DB 'Текст в окне зеленый',0
      CONT    DWORD ?
_DATA  ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH    0
      CALL    GetModuleHandleA@4
      MOV     [HINST], EAX
REG_CLASS:
;заполнить структуру окна
;стиль
      MOV     [WC.CLSSTYLE],stylcl
;процедура обработки сообщений
      MOV     [WC.CLSLPFNWNDPROC],OFFSET WNDPROC
      MOV     [WC.CLSCBCLSEXTRA],0
      MOV     [WC.CLSCBWNDEXTRA],0
      MOV     EAX,[HINST]
      MOV     [WC.CLSHINSTANCE],EAX
```

```
;----------пиктограмма окна
      PUSH    IDI_APPLICATION
      PUSH    0
      CALL    LoadIconA@8
      MOV     [WC.CLSHICON], EAX
;----------курсор окна
      PUSH    IDC_CROSS
      PUSH    0
      CALL    LoadCursorA@8
      MOV     [WC.CLSHCURSOR], EAX
;----------
      PUSH  RGBW                  ; цвет кисти
      CALL    CreateSolidBrush@4  ; создать кисть
      MOV     [WC.CLSHBRBACKGROUND],EAX
      MOV     DWORD PTR [WC.MENNAME],0
      MOV     DWORD PTR [WC.CLSNAME],OFFSET NAM
      PUSH    OFFSET WC
      CALL    RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH    0
      PUSH    [HINST]
      PUSH    0
      PUSH    0
      PUSH    DY0    ; DY0 - высота окна
      PUSH    DX0    ; DX0 - ширина окна
      PUSH    100    ; координата Y
      PUSH    100    ; координата X
      PUSH    WS_OVERLAPPEDWINDOW
      PUSH    OFFSET TITLENAME  ; имя окна
      PUSH    OFFSET NAM        ; имя класса
      PUSH    0
      CALL    CreateWindowExA@48
;проверка на ошибку
      CMP     EAX,0
      JZ     _ERR
      MOV    [NEWHWND], EAX   ; дескриптор окна
;---------------------------------
      PUSH    SW_SHOWNORMAL
      PUSH    [NEWHWND]
      CALL    ShowWindow@8    ; показать созданное окно
;---------------------------------
      PUSH    [NEWHWND]
      CALL    UpdateWindow@4  ; перерисовать видимую часть окна
;цикл обработки сообщений
MSG_LOOP:
      PUSH    0
```

```
      PUSH    0
      PUSH    0
      PUSH    OFFSET MSG
      CALL    GetMessageA@16
      CMP     AX, 0
      JE      END_LOOP
      PUSH    OFFSET MSG
      CALL    TranslateMessage@4
      PUSH    OFFSET MSG
      CALL    DispatchMessageA@4
      JMP     MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
      PUSH    MSG.MSWPARAM
      CALL    ExitProcess@4
_ERR:
      JMP    END_LOOP
;--------------------------------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    EBX
      PUSH    ESI
      PUSH    EDI
      CMP     DWORD PTR [EBP+0CH],WM_DESTROY
      JE      WMDESTROY
      CMP     DWORD PTR [EBP+0CH],WM_CREATE
      JE      WMCREATE
      CMP     DWORD PTR [EBP+0CH],WM_PAINT
      JE      WMPAINT
      JMP     DEFWNDPROC
WMPAINT:
;----------------
      PUSH    OFFSET PNT
      PUSH    DWORD PTR [EBP+08H]
      CALL    BeginPaint@8
      MOV     CONT,EAX       ; сохранить контекст (дескриптор)
;---------------- цвет фона = цвет окна
      PUSH    RGBW
      PUSH    EAX
      CALL    SetBkColor@8
```

```
;---------------- цвет текста (красный)
      PUSH    RGBT
      PUSH    CONT
      CALL    SetTextColor@8
;- вычислить длину текста в пикселах текста
      PUSH    OFFSET TEXT
      CALL    LENSTR
      PUSH    EBX    ; сохраним длину строки
      PUSH    OFFSET SZT
      PUSH    EBX
      PUSH    OFFSET TEXT
      PUSH    CONT
      CALL    GetTextExtentPoint32A@16
;---------------- размер окна
      PUSH    OFFSET RCT
      PUSH    DWORD PTR [EBP+8]
      CALL    GetWindowRect@8
;---------------- здесь вычисления координат
      MOV     EAX,RCT.R
      SUB     EAX,RCT.L
      SUB     EAX,SZT.X1
      SHR     EAX,1    ; текст посередине
      MOV     XT,EAX
      MOV     EAX,RCT.B
      SUB     EAX,RCT.T
      SHR     EAX,1
      SUB     EAX,25   ; учтем заголовочную часть окна
      MOV     YT,EAX
;---------------- вывести текст
;длина строки уже в стеке
      PUSH    OFFSET TEXT
      PUSH    YT
      PUSH    XT
      PUSH    CONT
      CALL    TextOutA@20
;---------------- закрыть контекст
      PUSH    OFFSET PNT
      PUSH    DWORD PTR [EBP+08H]
      CALL    EndPaint@8
      MOV     EAX, 0
      JMP     FINISH
WMCREATE:
      MOV     EAX, 0
      JMP     FINISH
DEFWNDPROC:
      PUSH    DWORD PTR [EBP+14H]
```

```
      PUSH    DWORD PTR [EBP+10H]
      PUSH    DWORD PTR [EBP+0CH]
      PUSH    DWORD PTR [EBP+08H]
      CALL    DefWindowProcA@16
      JMP     FINISH
WMDESTROY:
      PUSH    0
      CALL    PostQuitMessage@4 ;WM_QUIT
      MOV     EAX, 0
FINISH:
      POP     EDI
      POP     ESI
      POP     EBX
      POP     EBP
      RET     16
WNDPROC    ENDP
;длина строки
;указатель на строку в [EBP+08H]
LENSTR PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    ESI
      MOV     ESI,DWORD PTR [EBP+8]
      XOR     EBX,EBX
LBL1:
      CMP     BYTE PTR [ESI],0
      JZ      LBL2
      INC     EBX
      INC     ESI
      JMP     LBL1
LBL2:
      POP     ESI
      POP     EBP
      RET     4
LENSTR ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.1.2:

```
ml /c /coff  text2.asm
link /subsystem:windows text2.obj
```

Не могу не воспользоваться случаем и не восхититься теми возможностями, которые открывает перед программистом ассемблер. Вы можете передавать параметры через стек, а можете и через регистры. Если хотите — сохраняйте регистры в начале процедуры, а не хотите — не сохраняйте. Ассемблерный

код можно совершенствовать и еще раз совершенствовать. Кстати, для любителей цепочечных (строковых) команд далее привожу другую процедуру (листинг 2.1.3) определения длины строки, основанную на команде микропроцессора `SCAS`, позволяющей осуществлять поиск нужного элемента (байта — `SCASB`, слова — `SCASW`, двойного слова — `SCASD`) в строке. В качестве упражнения перепишите программу из листинга 2.1.2 с процедурой определения длины строки из листинга 2.1.3 и, откомпилировав, проверьте на работоспособность.

Листинг 2.1.3. Еще одна процедура определения длины строки

```
;длина строки - [EBP+08H]
LENSTR PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    EAX
      PUSH    EDI
;--------------------
      CLD
      MOV     EDI,DWORD PTR [EBP+08H]
      MOV     EBX,EDI
      MOV     ECX,100    ; ограничить длину строки
      XOR     AL,AL
      REPNE   SCASB      ; найти символ 0
      SUB     EDI,EBX    ; длина строки, включая 0
      MOV     EBX,EDI
      DEC     EBX        ; теперь здесь длина строки
;--------------------
      POP EDI
      POP EAX
      POP EBP
      RET 4
LENSTR ENDP
```

Выбор шрифта

Рассмотрим теперь вопрос о том, как выводить текстовую информацию при помощи различных шрифтов. Удобнее всего задать свойства шрифта посредством функции `CreateFontIndirect`, параметром которой является указатель на структуру `LOGFONT`. Хотя название функции и начинается со слова "Create", речь идет не о создании, а, скорее, об изменении существующего шрифта согласно заданным параметрам. Существует и другая функция — `CreateFont`, в которой свойства шрифта задаются параметрами функции (а не полями

структуры). На мой взгляд, она менее удобна при использовании на ассемблере — поработайте с ней сами, если хотите. Выбор нужного шрифта осуществляется функцией `SelectObject`.

Начнем с того, что разберем поля этой структуры.

```
LOGFONT STRUC
      LfHeight          DD ?
      LfWidth           DD ?
      LfEscapement      DD ?
      LfOrientation     DD ?
      LfWeight          DD ?
      LfItalic          DB ?
      LfUnderline       DB ?
      LfStrikeOut       DB ?
      LfCharSet         DB ?
      LfOutPrecision    DB ?
      LfClipPrecision   DB ?
      LfQuality         DB ?
      LfPitchAndFamily  DB ?
      LfFaceName        DB 32 DUP(0)
LOGFONT ENDS
```

Здесь:

- `LfHeight` — определяет высоту шрифта в логических единицах; если 0, то высота берется по умолчанию;
- `LfWidth` — определяет ширину шрифта в логических единицах; если 0, то ширина по умолчанию;
- `LfEscapement` — угол наклона текста в десятых долях градуса по отношению к горизонтальной оси в направлении против часовой стрелки;
- `LfOrientation` — то же, что и предыдущий параметр, но по отношению к отдельному символу;
- `LfWeight` — задает жирность шрифта (0—900);
- `LfItalic` — если 1, то курсив;
- `LfUnderline` — если 1, то символы подчеркнуты;
- `LfStrikeOut` — если 1, то символы перечеркнуты;
- `LfCharSet` — задает множество символов шрифта, обычно определяется константой `ANSI_CHARSET` (0);
- `LfOutPrecision` — флаг точности шрифта; определяет, насколько точно созданный шрифт отвечает заданным параметрам. Возможные значения:
  - `OUT_DEFAULT_PRECIS = 0`;
  - `OUT_STRING_PRECIS = 1`;

- `OUT_CHARACTER_PRECIS = 2`;
- `OUT_STROKE_PRECIS = 3`;
- `OUT_TT_PRECIS = 4`;
- `OUT_DEVICE_PRECIS = 5`;
- `OUT_RASTER_PRECIS = 6`;
- `OUT_TT_ONLY_PRECIS = 7`;
- `OUT_OUTLINE_PRECIS = 8`;
- `OUT_SCREEN_OUTLINE_PRECIS = 9`;

❒ `LfClipPrecision` — флаг точности прилегания шрифта; определяет, как будут отсекаться части шрифта, не попадающие в видимую область. Возможные значения:

- `CLIP_DEFAULT_PRECIS = 0`;
- `CLIP_CHARACTER_PRECIS = 1`;
- `CLIP_STROKE_PRECIS = 2`;
- `CLIP_MASK = 0fH`;
- `CLIP_LH_ANGLES = (1 SHL 4)`;
- `CLIP_TT_ALWAYS = (2 SHL 4)`;
- `CLIP_EMBEDDED = (8 SHL 4)`;

❒ `LfQuality` — флаг качества шрифта; определяет соответствие логического шрифта и шрифта, допустимого данным устройством. Возможные значения:

- `DEFAULT_QUALITY = 0`;
- `DRAFT_QUALITY = 1`;
- `PROOF_QUALITY = 2`;

❒ `LfPitchAndFamily` — определяет тип и семейство шрифта. Возможные значения определяются комбинацией (ИЛИ) двух групп констант:

- `DEFAULT_PITCH = 0`;
- `FIXED_PITCH = 1`;
- `VARIABLE_PITCH = 2`

и

- `FF_DONTCARE = 0`;
- `FF_ROMAN = (1 SHL 4)`;
- `FF_SWISS = (2 SHL 4)`;

- `FF_MODERN = (3 SHL 4)`;
- `FF_SCRIPT = (4 SHL 4)`;
- `FF_DECORATIVE = (5 SHL 4)`;

- `LfFaceName` — содержит название шрифта. Длина имени не может превосходить 32-х символов.

Обратимся к примеру задания своего шрифта (результат работы программы — на рис. 2.1.1). Однако поскольку большая часть программы будет совпадать с аналогичной частью предыдущих программ, я приведу здесь только необходимые фрагменты. Рассмотрим сначала фрагмент, выполняющийся при получении сообщения `WM_PAINT` (листинг 2.1.4).

Листинг 2.1.4. Фрагмент программы, выводящей текст с заданным шрифтом

```
WMPAINT:
;------- определить контекст
      PUSH    OFFSET PNT
      PUSH    DWORD PTR [EBP+08H]
      CALL    BeginPaint@8
      MOV     CONT,EAX ;сохранить контекст (дескриптор)
;------- цвет фона = цвет окна
      PUSH    RGBW
      PUSH    EAX
      CALL    SetBkColor@8
;------- цвет текста (красный)
      PUSH    RGBT
      PUSH    CONT
      CALL    SetTextColor@8
;------ здесь определение координат
      MOV     XT,150
      MOV     YT,120
;------ задать (создать) шрифт
      MOV     lg.lfHeight,14         ; высота шрифта
      MOV     lg.lfWidth,8           ; ширина шрифта
      MOV     lg.lfEscapement,900    ; ориентация
      MOV     lg.lfOrientation,0     ; вертикальная
      MOV     lg.lfWeight,0          ; толщина линий шрифта
      MOV     lg.lfItalic,0          ; курсив
      MOV     lg.lfUnderline,0       ; подчеркивание
      MOV     lg.lfStrikeOut,0       ; перечеркивание
      MOV     lg.lfCharSet,204       ; набор шрифтов RUSSIAN_CHARSET
      MOV     lg.lfOutPrecision,0
      MOV     lg.lfClipPrecision,0
      MOV     lg.lfQuality,1
```

```
      MOV     lg.lfPitchAndFamily,0
      PUSH    OFFSET lg
;задать название шрифта
      PUSH    OFFSET NFONT
      PUSH    OFFSET lg.LfFaceName
      CALL    COPYSTR
      CALL    CreateFontIndirectA@4
;------ выбрать созданный объект
      PUSH    EAX
      PUSH    CONT
      CALL    SelectObject@8
      PUSH    EAX
;------ вычислить длину текста в пикселах текста
      PUSH    OFFSET TEXT
      CALL    LENSTR
;------------- вывести текст ----------------
      PUSH    EBX
      PUSH    OFFSET TEXT
      PUSH    YT
      PUSH    XT
      PUSH    CONT
      CALL    TextOutA@20
;удалить объект "FONT"
;идентификатор уже в стеке
      CALL    DeleteObject@4
;---------------- закрыть контекст
      PUSH    OFFSET PNT
      PUSH    DWORD PTR [EBP+08H]
      CALL    EndPaint@8
      MOV     EAX, 0
      JMP     FINISH
```

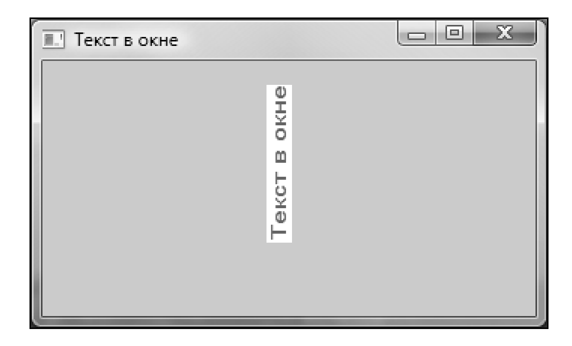


Рис. 2.1.1. Вывод текста под углом 90°

Как видно из фрагмента, создание шрифта производится по следующей схеме: надо задать шрифт при помощи функции `CreateFontIndirect`, выбрать шрифт функцией `SelectObject`, вывести текст заданным шрифтом, удалить созданный шрифт (объект). Поле `LfFaceName` структуры `LOGFONT` должно содержать название шрифта. Если такого шрифта нет, текст выводится шрифтом по умолчанию. Название шрифта у нас задано в строке `NFONT`, и мы копируем его в поле `LfFaceName` при помощи функции `COPYSTR`, текст которой приводится в листинге 2.1.5. Разумеется, строка `NFONT` должна быть задана в сегменте данных и содержать название шрифта, например `'Arial'` или `'Courier New'`. Кроме этого, не забудьте, что `LG` — это структура типа `LOGFONT`, которую мы разбирали в начале этого раздела. На компакт-диске, который прилагается к книге, эта программа содержится полностью.

Листинг 2.1.5. Процедура копирования одной строки в другую

```
;процедура копирования одной строки в другую
;строка, куда копировать [EBP+08H]
;строка, что копировать [EBP+0CH]
COPYSTR  PROC
      PUSH EBP
      MOV  EBP,ESP
      MOV  ESI,DWORD PTR [EBP+0CH]
      MOV  EDI,DWORD PTR [EBP+08H]
L1:
      MOV  AL,BYTE PTR [ESI]
      MOV  BYTE PTR [EDI],AL
      CMP  AL,0
      JE   L2
      INC  ESI
      INC  EDI
      JMP  L1
L2:
      POP  EBP
      RET  8
COPYSTR ENDP
```

Процедура из листинга 2.1.5 не учитывает длину строки, в которую производится копирование. Лучше всего это учесть, включив еще один параметр — максимальное количество символов, которое может быть скопировано. Попробуйте это сделать самостоятельно.

В заключение раздела мы рассмотрим один очень важный вопрос. При разборе предыдущих примеров этот вопрос, скорее всего, у вас не возникал, и вот почему. Весь вывод информации происходил в программе при получении

сообщения `WM_PAINT`. В реальных программах вывод информации в окно может происходить по различным событиям и из различных процедур. Кроме того, если информации в окне много, то непосредственный вывод при помощи функции `TextOut` довольно медленный. Чтобы воспроизводить содержимое окна, необходимо где-то запомнить это содержимое. Возникает проблема сохранения информации (и не только текстовой), находящейся в окне.

Если кто-то программировал для операционной системы MS-DOS, то знает, что там подобная проблема также возникает. Решается она следующим образом: используется фоновая видеостраница, на которую выводится вся информация. Затем фоновая страница копируется на видимую страницу. При этом создается впечатление, что информация появляется на экране мгновенно. В качестве фоновой страницы может быть использована как область ОЗУ, так и область видеопамяти.

Аналогично в операционной системе Windows образуется виртуальное окно, и весь вывод информации производится туда. Затем по приходе сообщения `WM_PAINT` содержимое виртуального окна копируется на реальное окно. В целом общая схема такова.

- При создании окна:
  - создается совместимый контекст устройства. Для этого используется функция `CreateCompatibleDC`. Полученный контекст (дескриптор) следует запомнить;
  - создается карта битов, совместимая с данным контекстом. Для этой цели применяется функция `CreateCompatibleBitmap`;
  - выбирается кисть цветом, совпадающим с цветом основного окна;
  - создается битовый шаблон путем выполнения растровой операции с использованием выбранной кисти. Выполняется с помощью функции `PatBlt`.
- Вся информация выводится в виртуальное окно и дается команда перерисовки окна. Действия выполняются функцией `InvalidateRect`.
- При получении сообщения `WM_PAINT` содержимое виртуального окна копируется на реальное окно посредством функции `BitBlt`.

Изложенная схема будет применена на практике в следующем разделе.

Графические образы

Этот раздел главы посвящается графике. Впрочем, основы графики в Windows, в принципе, достаточно тривиальны, поэтому мы рассмотрим один

простой пример — вывод графических образов. Но вначале я изложу некоторые опорные моменты.

- Система координат для вывода графических образов такая же, как и для ввода текстовой информации. Координаты измеряются в логических единицах, которые по умолчанию совпадают с пикселами. При желании эту пропорцию можно изменить.
- Цвет рисования образуется тремя способами. При использовании функции `SetPixel` задается цвет данной точки. Для линий необходимо задать цвет пера. Для задания цвета замкнутых графических объектов следует задать цвет кисти.
- Перо создается при помощи функции `CreatePen`, кисть — при помощи функции `CreateSolidBrush` (мы ее уже использовали). Для создания разноцветной картинки можно заранее создать несколько кистей и перьев, а затем в нужный момент выбирать при помощи функции `SelectObject` (мы также уже использовали эту функцию).
- Для рисования можно использовать следующие функции API:
  - `SetPixel` — установить заданный цвет пиксела;
  - `LineTo` — провести линию от текущей точки до точки с заданными координатами, которая в свою очередь становится текущей;
  - `MoveToEx` — сменить текущую точку;
  - `Arc` — рисование дуги;
  - `Rectangle` — рисование прямоугольника;
  - `RoundRect` — рисование прямоугольника со скругленными углами;
  - `Ellipse`, `Pie` — рисование эллипсов и секторов.
- Если при рисовании замкнутой фигуры был установлен цвет кисти, отличный от цвета основного фона, то замкнутая фигура окрашивается этим цветом.
- Для установки соотношения между логическими единицами и пикселами используется функция `SetMapMode`.
- Можно установить область вывода при помощи функции `SetViewportExtEx`. Посредством функции `SetViewportOrgEx` можно задать начало области ввода.

После всего сказанного пора продемонстрировать программу. Программа достаточно проста, но в ней заложены основы работы с графикой. По щелчку левой кнопки мыши сначала появляется горизонтальная линия, по второму щелчку — наклонная линия, по третьему щелчку — заполненный прямо-

угольник. Программа представлена в листинге 2.1.6, результат ее работы — на рис. 2.1.2. Обратите внимание, что при создании окна (сообщение `WM_CREATE`) одновременно создается и виртуальное окно. По щелчку мыши (сообщение `WM_LBUTTONDOWN`) чертятся геометрические фигуры, которые вначале заносятся в виртуальное окно. А затем принудительно посылается сообщение `WM_PAINT` (функция `InvalidateRect`), и тогда виртуальное окно копируется на реальное окно.

Листинг 2.1.6. Простая программа для демонстрации графики

```
;файл graph1.inc
;константы
;сообщение приходит при закрытии окна
WM_DESTROY    equ 2
;сообщение приходит при создании окна
WM_CREATE     equ 1
;сообщение при щелчке левой кнопкой мыши в области окна
WM_LBUTTONDOWN equ 201h
;сообщение приходит при перерисовке окна
WM_PAINT      equ 0Fh
;свойства окна
CS_VREDRAW    equ 1h
CS_HREDRAW    equ 2h
CS_GLOBALCLASS equ 4000h
WS_OVERLAPPEDWINDOW equ 000CF0000H
stylcl        equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
DX0           equ 600
DY0           equ 400
;компоненты цветов
RGBW          equ (50 or (50 shl 8)) or (255 shl 16) ; цвет окна
RGBR          equ 150 ; цвет региона
RGBL          equ 0   ; цвет линии
RGBP          equ 255 or (100 shl 8) ; цвет точки
;идентификатор стандартной пиктограммы
IDI_APPLICATION  equ 32512
;идентификатор курсора
IDC_CROSS        equ 32515
;режим показа окна - нормальный
SW_SHOWNORMAL    equ 1
;прототипы внешних процедур
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
```

```
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
EXTERN  BeginPaint@8:NEAR
EXTERN  EndPaint@8:NEAR
EXTERN  GetStockObject@4:NEAR
EXTERN  CreateSolidBrush@4:NEAR
EXTERN  GetSystemMetrics@4:NEAR
EXTERN  GetDC@4:NEAR
EXTERN  CreateCompatibleDC@4:NEAR
EXTERN  SelectObject@8:NEAR
EXTERN  CreateCompatibleBitmap@12:NEAR
EXTERN  PatBlt@24:NEAR
EXTERN  BitBlt@36:NEAR
EXTERN  ReleaseDC@8:NEAR
EXTERN  DeleteObject@4:NEAR
EXTERN  InvalidateRect@12:NEAR
EXTERN  GetStockObject@4:NEAR
EXTERN  DeleteDC@4:NEAR
EXTERN  CreatePen@12:NEAR
EXTERN  SetPixel@16:NEAR
EXTERN  LineTo@12:NEAR
EXTERN  MoveToEx@16:NEAR
EXTERN  Rectangle@20:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND          DD ?   ; идентификатор окна, получающего сообщение
      MSMESSAGE       DD ?   ; идентификатор сообщения
      MSWPARAM        DD ?   ; дополнительная информация о сообщении
      MSLPARAM        DD ?   ; дополнительная информация о сообщении
      MSTIME          DD ?   ; время посылки сообщения
      MSPT            DD ?   ; положение курсора во время
                             ; посылки сообщения
MSGSTRUCT ENDS
;---------
WNDCLASS STRUC
      CLSSTYLE        DD ?   ; стиль окна
      CLSLPFNWNDPROC  DD ?   ; указатель на процедуру окна
      CLSCBCLSEXTRA   DD ?   ; информация о дополнительных байтах
```

```
                              ; для данной структуры
      CLSCBWNDEXTRA  DD ?   ; информация о дополнительных байтах для окна
      CLSHINSTANCE   DD ?   ; дескриптор приложения
      CLSHICON       DD ?   ; идентификатор пиктограммы окна
      CLSHCURSOR     DD ?   ; идентификатор курсора окна
      CLSHBRBACKGROUND DD ? ; идентификатор кисти окна
      MENNAME        DD ?   ; имя-идентификатор меню
      CLSNAME        DD ?   ; специфицирует имя класса окон
WNDCLASS ENDS
;---
PAINTSTR STRUC
      Hdc     DD 0
      fErase  DD 0
      left    DD 0
      top     DD 0
      right   DD 0
      bottom  DD 0
      fRes    DD 0
      fIncUp  DD 0
      Reserv  DB 32 dup(0)
PAINTSTR ENDS
;----
RECT STRUC
      L    DD ?   ; X-координата левого верхнего угла
      T    DD ?   ; Y-координата левого верхнего угла
      R    DD ?   ; X-координата правого нижнего угла
      B    DD ?   ; Y-координата правого нижнего угла
RECT ENDS
;файл graph.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;-------------------------------------------------
include graph1.inc
;подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\gdi32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      NEWHWND DWORD 0
      MSG     MSGSTRUCT <?>
      WC      WNDCLASS  <?>
      PNT     PAINTSTR  <?>
      HINST   DWORD 0
```

```
      TITLENAME BYTE 'Графика в окне',0
      NAM     BYTE 'CLASS32',0
      XT      DWORD 30
      YT      DWORD 30
      XM      DWORD ?
      YM      DWORD ?
      HDC     DWORD ?
      MEMDC   DWORD ?
      HPEN    DWORD ?
      HBRUSH  DWORD ?
      P       DWORD 0   ; признак вывода
      XP      DWORD ?
      YP      DWORD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH    0
      CALL    GetModuleHandleA@4
      MOV     HINST, EAX
REG_CLASS:
;заполнить структуру окна
;стиль
      MOV     WC.CLSSTYLE,stylcl
;процедура обработки сообщений
      MOV     WC.CLSLPFNWNDPROC,OFFSET WNDPROC
      MOV     WC.CLSCBCLSEXTRA,0
      MOV     WC.CLSCBWNDEXTRA,0
      MOV     EAX,HINST
      MOV     WC.CLSHINSTANCE,EAX
;----------пиктограмма окна
      PUSH    IDI_APPLICATION
      PUSH    0
      CALL    LoadIconA@8
      MOV     [WC.CLSHICON], EAX
;----------курсор окна
      PUSH    IDC_CROSS
      PUSH    0
      CALL    LoadCursorA@8
      MOV     WC.CLSHCURSOR, EAX
;----------
      PUSH    RGBW ;цвет кисти
      CALL    CreateSolidBrush@4 ;создать кисть
      MOV     WC.CLSHBRBACKGROUND,EAX
      MOV     DWORD PTR WC.MENNAME,0
```

```
      MOV     DWORD PTR WC.CLSNAME,OFFSET NAM
      PUSH    OFFSET WC
      CALL    RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH    0
      PUSH    [HINST]
      PUSH    0
      PUSH    0
      PUSH    DY0    ; DY0 - высота окна
      PUSH    DX0    ; DX0 - ширина окна
      PUSH    100    ; координата Y
      PUSH    100    ; координата X
      PUSH    WS_OVERLAPPEDWINDOW
      PUSH    OFFSET TITLENAME    ; имя окна
      PUSH    OFFSET NAM          ; имя класса
      PUSH    0
      CALL    CreateWindowExA@48
;проверка на ошибку
      CMP     EAX,0
      JZ     _ERR
      MOV     NEWHWND, EAX        ; дескриптор окна
;-----------------------------------
      PUSH    SW_SHOWNORMAL
      PUSH    NEWHWND
      CALL    ShowWindow@8        ; показать созданное окно
;-----------------------------------
      PUSH    NEWHWND
      CALL    UpdateWindow@4      ; перерисовать видимую часть окна
;цикл обработки сообщений
MSG_LOOP:
      PUSH    0
      PUSH    0
      PUSH    0
      PUSH    OFFSET MSG
      CALL    GetMessageA@16
      CMP     AX, 0
      JE      END_LOOP
      PUSH    OFFSET MSG
      CALL    TranslateMessage@4
      PUSH    OFFSET MSG
      CALL    DispatchMessageA@4
      JMP     MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
      PUSH    MSG.MSWPARAM
      CALL    ExitProcess@4
```

```
_ERR:
      JMP     END_LOOP
;-----------------------------------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH    EBP
      MOV     EBP,ESP
      PUSH    EBX
      PUSH    ESI
      PUSH    EDI
      CMP     DWORD PTR [EBP+0CH],WM_DESTROY
      JE      WMDESTROY
      CMP     DWORD PTR [EBP+0CH],WM_CREATE
      JE      WMCREATE
      CMP     DWORD PTR [EBP+0CH],WM_PAINT
      JE      WMPAINT
      CMP     DWORD PTR [EBP+0CH],WM_LBUTTONDOWN
      JE      LBUTTON
      JMP     DEFWNDPROC
LBUTTON:
      CMP     P,0
      JNE     F1
;линия точками (горизонтальная)
      MOV     YP,50 ;Y
      MOV     XP,10 ;X
      MOV     ECX,200
LL:
      PUSH    ECX
      PUSH    RGBP
      PUSH    YP
      PUSH    XP
      PUSH    MEMDC
      CALL    SetPixel@16
      INC     XP
      POP     ECX
      LOOP    LL
      INC     P
      JMP     F3
F1:
      CMP     P,1
      JNE     F2
```

```
;вначале установим текущие координаты на конец предыдущей линии
      PUSH    0
      PUSH    YP
      PUSH    XP
      PUSH    MEMDC
      CALL    MoveToEx@16
;линия пером
      PUSH    300
      PUSH    550
      PUSH    MEMDC
      CALL    LineTo@12
      INC     P
      JMP     F3
F2:
      CMP     P,2
      JNE     FIN
;замкнутая фигура - прямоугольник
;вначале выбрать кисть для заполнения области
      PUSH    HBRUSH
      PUSH    MEMDC
      CALL    SelectObject@8
;теперь рисуем заполненный прямоугольник
;если не выбирать кисть, то будет
;нарисован незаполненный прямоугольник
      PUSH    350
      PUSH    400
      PUSH    200
      PUSH    200
      PUSH    MEMDC
      CALL    Rectangle@20
      INC     P
F3:
;дать команду перерисовать окно
      PUSH    0
      PUSH    OFFSET RECT
      PUSH    DWORD PTR [EBP+08H]
      CALL    InvalidateRect@12
FIN:
      MOV     EAX, 0
      JMP     FINISH
WMPAINT:
      PUSH    OFFSET PNT
      PUSH    DWORD PTR [EBP+08H]
      CALL    BeginPaint@8
      MOV     HDC,EAX   ; сохранить контекст (дескриптор)
```

```
;скопировать виртуальное окно на реальное
      PUSH    0CC0020h ;SRCCOPY=изображение как есть
      PUSH    0       ; y - источника
      PUSH    0       ; x - источника
      PUSH    MEMDC   ; контекст источника
      PUSH    YM      ; высота - куда
      PUSH    XM      ; ширина - куда
      PUSH    0       ; y - куда
      PUSH    0       ; x - куда
      PUSH    HDC     ; контекст - куда
      CALL    BitBlt@36
;---------------- закрыть контекст окна
      PUSH    OFFSET PNT
      PUSH    DWORD PTR [EBP+08H]
      CALL    EndPaint@8
      MOV     EAX, 0
      JMP     FINISH
WMCREATE:
;размеры экрана
      PUSH    0 ;X
      CALL    GetSystemMetrics@4
      MOV     XM,EAX
      PUSH    1 ;Y
      CALL    GetSystemMetrics@4
      MOV     YM,EAX
;открыть контекст окна
      PUSH    DWORD PTR [EBP+08H]
      CALL    GetDC@4
      MOV     HDC,EAX
;создать совместимый с данным окном контекст
      PUSH    EAX
      CALL    CreateCompatibleDC@4
      MOV     MEMDC,EAX
;создать в памяти растровое изображение, совместимое с hdc
      PUSH    YM
      PUSH    XM
      PUSH    HDC
      CALL    CreateCompatibleBitmap@12
;выбрать растровое изображение в данном контексте
      PUSH    EAX
      PUSH    MEMDC
      CALL    SelectObject@8
;цвет кисти
      PUSH    RGBW
      CALL    CreateSolidBrush@4   ; создать кисть
```

```
;выбрать кисть в данном контексте
      PUSH    EAX
      PUSH    MEMDC
      CALL    SelectObject@8
;заполнить данную прямоугольную область
      PUSH    0F00021h ;PATCOPY=заполнить данным цветом
      PUSH    YM
      PUSH    XM
      PUSH    0
      PUSH    0
      PUSH    MEMDC
      CALL    PatBlt@24
;создать кисть и перо для рисования
;цвет кисти
      PUSH    RGBR
      CALL    CreateSolidBrush@4 ;создать кисть
      MOV     HBRUSH,EAX
;задать перо
      PUSH    RGBR    ; цвет
      PUSH    0       ; толщина=1
      PUSH    0       ; сплошная линия
      CALL    CreatePen@12
      MOV     HPEN,EAX
;удалить контекст
      PUSH    HDC
      PUSH    DWORD PTR [EBP+08H]
      CALL    ReleaseDC@8
      MOV     EAX, 0
      JMP     FINISH
DEFWNDPROC:
      PUSH    DWORD PTR [EBP+14H]
      PUSH    DWORD PTR [EBP+10H]
      PUSH    DWORD PTR [EBP+0CH]
      PUSH    DWORD PTR [EBP+08H]
      CALL    DefWindowProcA@16
      JMP     FINISH
WMDESTROY:
;удалить перо
      PUSH    HPEN
      CALL    DeleteDC@4
;удалить кисть
      PUSH    HBRUSH
      CALL    DeleteDC@4
;удалить виртуальное окно
      PUSH    MEMDC
      CALL    DeleteDC@4
```

```
;выход
      PUSH    0
      CALL    PostQuitMessage@4 ;WM_QUIT
      MOV     EAX, 0
FINISH:
      POP     EDI
      POP     ESI
      POP     EBX
      POP     EBP
      RET     16
WNDPROC  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.1.6:

```
ml /c /coff  graph.asm
link /subsystem:windows graph.obj
```

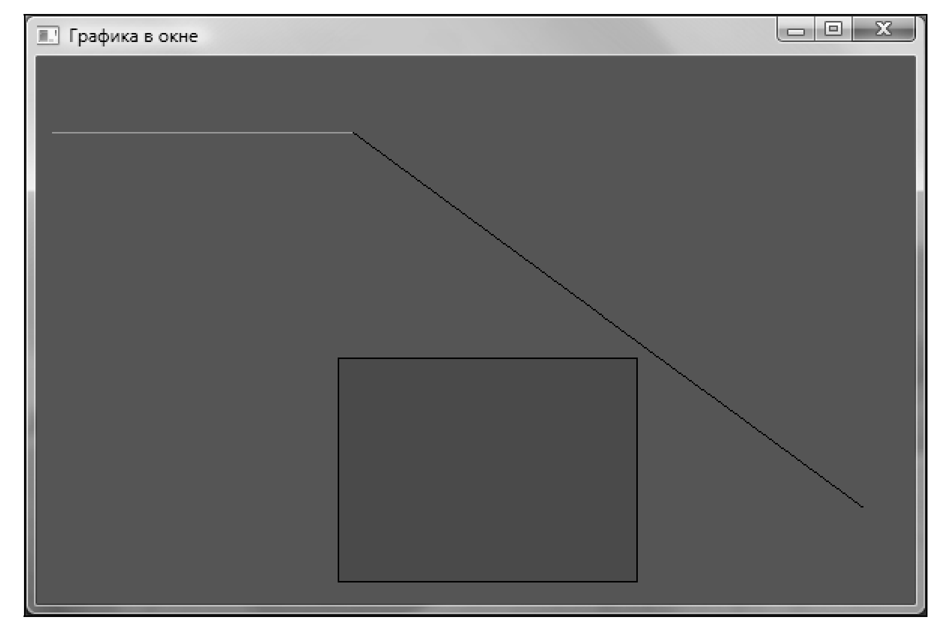


Рис. 2.1.2. Результат работы программы из листинга 2.1.6

Рассказ о графическом выводе будет неполным, если не коснуться вопроса о манипулировании растровыми изображениями. Рассмотрим последователь-

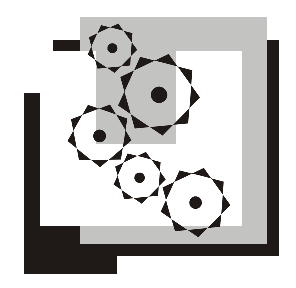
ность действий, которые необходимо выполнить для вывода растрового изображения (примеры вывода будут даны в последующих главах):

1. Загрузить растровое изображение и запомнить его дескриптор.
2. Получить контекст устройства для области памяти, где будет храниться изображение.
3. Выбрать изображение в данном контексте.
4. Скопировать изображение на экран (функция `BitBlt`).

С растровыми изображениями удобно работать при помощи ресурсов, но их можно создавать непосредственно в программе или считывать из файла. Но об этом позднее. С графикой же мы продолжим работать в следующей главе.

Замечание

Специалисту, знакомому с программированием в операционной системе Windows, возможно, покажется странным, что мы пишем в данной главе собственные строковые функции, вместо того чтобы воспользоваться существующими в Windows соответствующими API-функциями. Да-да, такие функции существуют, дорогой читатель. Причина моего пренебрежения этими функциями проста. Во-первых, я рассчитываю не только на "продвинутых" читателей, но и на людей, обучающихся программированию на ассемблере. Во-вторых, как я уже говорил во *введении*, данная книга — это попытка создания некоторого симбиоза ассемблера и программирования в Windows. Следуя этому принципу, мы не всегда будем решать задачи только средствами API-функций. Однако, понимая всю важность строковых API-функций, в свое время я приведу примеры их использования. Кроме того, в *приложении 1* будет дано их полное описание.



Глава 2.2

Графика: GDI+, DirectX, OpenGL

Кроме стандартной системной библиотеки GDI, которую мы использовали в предыдущей главе, начиная с Windows XP, фирма Microsoft поставляет улучшенный вариант графической библиотеки GDI+. Помимо этого, Microsoft поддерживает мощную библиотеку DirectX, содержащую не только графические, но также и мультимедийные возможности. Еще одна графическая библиотека — OpenGL — сторонних разработчиков (Silicon Graphics Inc.) также включается в поставку Windows. Обо всем этом мы будем говорить в данной главе.

Работаем с функциями GDI+

В *главе 2.1* мы довольно подробно остановились на выводе графической информации при помощи функций GDI. Эта библиотека существует в Windows еще с середины 80-х годов прошлого века. Библиотека GDI+ — это новая библиотека, которая пришла на смену GDI. Она присутствует в операционных системах Windows XP, Windows Server 2003 и вот теперь, разумеется, в Windows Vista. Приложения, написанные с использованием библиотеки GDI, будут, конечно, работать, но новые приложения следует писать, опираясь уже на GDI+. Основной особенностью новой графической подсистемы является значительно более широкий перечень графических возможностей по отношению к старой GDI. Например, в библиотеку помещены возможности координатных преобразований (сплайны, кривые Безье, поворот графических объектов и др.). В библиотеке реализованы, в частности, такие эффекты, как прозрачность и сглаживание. Наконец, значительно расширен перечень графических форматов файлов, с которыми работают библиотечные функции. Появилась, например, возможность отображать файлы формата GIF с анимацией.

Основным модулем, который осуществляет графические функции GDI+, является динамическая библиотека gdiplus.dll. Для использования ее в программах на языке ассемблера необходима библиотека импорта gdiplus.lib, которую можно найти, например, в пакете Visual Studio .NET[1]. Описание почти всех функций GDI+ можно получить в справочнике MSDN (см. сайт Microsoft: **http://msdn.microsoft.com**).

> ***Замечание***
>
> В *главе 3.3* мы узнаем, что доступ к ресурсам (функциям и данным) динамических библиотек можно получить и без применения библиотеки импорта. Для этого используется так называемое явное динамическое связывание посредством API-функции `LoadLibrary`.

Библиотека GDI+ — это объектная библиотека, чего мы ни в коей мере не будем касаться в нашем рассмотрении. Наша задача более проста — показать, как в программе на языке ассемблера, не принимая во внимание объектную сущность библиотеки, писать графические программы, опираясь на новые мощные возможности.

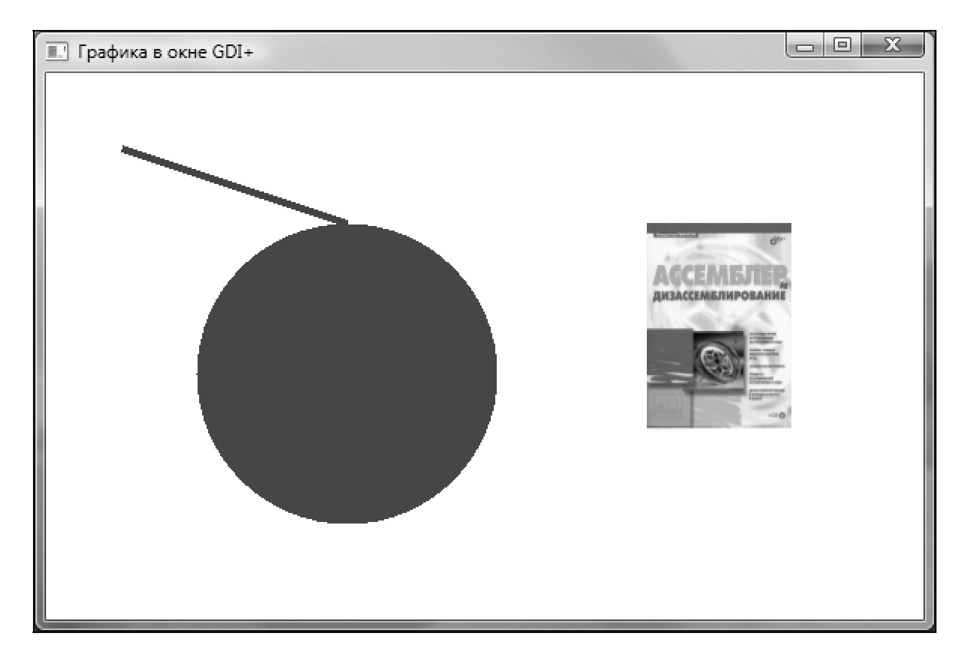


Рис. 2.2.1. Пример окна с графикой GDI+

[1] В программе из листинга 2.2.1 я использовал библиотеку gdiplus.lib именно из пакета Visual Studio .NET 2005.

В листинге 2.2.1 представлен простой пример использования библиотеки GDI+. По щелчку мыши в окне появляются графические объекты: линия, закрашенный круг, графическое изображение, хранящееся в файле (рис. 2.2.1). В *главе 2.1* мы узнали, как сохранять графические объекты при перерисовке окна. Считая, что читатель хорошо разобрался в этом механизме, я исключил его из программы, дабы не перегружать код. Я также настоятельно рекомендую читателю освежить в памяти материал *главы 1.5*, где мы в частности говорили о кодировке Unicode. Библиотека GDI+ полностью функционирует в этой кодировке.

Листинг 2.2.1. Пример использования функций GDI+

```
;файл graph2.inc
;константы
;сообщение приходит при закрытии окна
WM_DESTROY    equ 2
;сообщение приходит при создании окна
WM_CREATE     equ 1
;сообщение приходит при перерисовке окна
WM_PAINT      equ 0Fh
;сообщение при щелчке левой кнопкой мыши в области окна
WM_LBUTTONDOWN equ 201h
;свойства окна
CS_VREDRAW     equ 1h
CS_HREDRAW     equ 2h
CS_GLOBALCLASS equ 4000h
WS_OVERLAPPEDWINDOW equ 000CF0000H
stylcl  equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
DX0            equ 600
DY0            equ 400
;цвета
RED            equ 255
GREEN          equ 255
BLUE           equ 255
RGBW           equ ((RED or (GREEN shl 8)) or (BLUE shl 16))
;идентификатор стандартной пиктограммы
IDI_APPLICATION  equ 32512
;идентификатор курсора
IDC_CROSS      equ 32515
;режим показа окна - нормальный
SW_SHOWNORMAL  equ 1
;прототипы внешних процедур
EXTERN  GdipDrawImageI@16:NEAR
EXTERN  GdipDisposeImage@4:NEAR
EXTERN  GdipLoadImageFromFile@8:NEAR
```

```
EXTERN  GdipFillEllipseI@24:NEAR
EXTERN  GdipDeleteBrush@4:NEAR
EXTERN  GdipCreateSolidFill@8:NEAR
EXTERN  GetDC@4:NEAR
EXTERN  ReleaseDC@8:NEAR
EXTERN  GdipDrawLineI@24:NEAR
EXTERN  GdipDeletePen@4:NEAR
EXTERN  GdipCreatePen1@16:NEAR
EXTERN  GdipDeleteGraphics@4:NEAR
EXTERN  GdipCreateFromHDC@8:NEAR
EXTERN  GdiplusShutdown@4:NEAR
EXTERN  GdiplusStartup@12:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  CreateSolidBrush@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
EXTERN  BeginPaint@8:NEAR
EXTERN  EndPaint@8:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?   ; идентификатор окна, получающего сообщение
      MSMESSAGE  DD ?   ; идентификатор сообщения
      MSWPARAM   DD ?   ; дополнительная информация о сообщении
      MSLPARAM   DD ?   ; дополнительная информация о сообщении
      MSTIME     DD ?   ; время посылки сообщения
      MSPT       DD ?   ; положение курсора во время
                        ; посылки сообщения
MSGSTRUCT ENDS
;---------
WNDCLASS STRUC
      CLSSTYLE       DD ?   ; стиль окна
      CLSLPFNWNDPROC DD ?   ; указатель на процедуру окна
      CLSCBCLSEXTRA  DD ?   ; информация о дополнительных байтах
                            ; для данной структуры
      CLSCBWNDEXTRA  DD ?   ; информация о дополнительных байтах для окна
```

```
      CLSHINSTANCE   DD ?   ; дескриптор приложения
      CLSHICON       DD ?   ; идентификатор пиктограммы окна
      CLSHCURSOR     DD ?   ; идентификатор курсора окна
      CLSHBRBACKGROUND DD ? ; идентификатор кисти окна
      MENNAME        DD ?   ; имя-идентификатор меню
      CLSNAME        DD ?   ; специфицирует имя класса окон
WNDCLASS ENDS
;---
PAINTSTR STRUC
      Hdc     DD 0
      fErase  DD 0
      left    DD 0
      top     DD 0
      right   DD 0
      bottom  DD 0
      fRes    DD 0
      fIncUp  DD 0
      Reserv  DB 32 dup(0)
PAINTSTR ENDS
;файл graph2.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;-------------------------------------------------
include graph2.inc
;подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\gdi32.lib
includelib c:\masm32\lib\gdiplus.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      TEXT1   DB '  ', 0
      TEXT2   DB 'Сообщение', 0
      NEWHWND DWORD 0
      MSG     MSGSTRUCT <?>
      WC      WNDCLASS  <?>
      PNT     PAINTSTR  <?>
      HINST   DD 0
      TITLENAME DB 'Графика в окне GDI+',0
      NAM     DB 'CLASS32',0
      HDC     DD ?
      GDIPLUS1 DD 1
               DD 0
               DD 0
               DD 0
```

```
      GDIPLUS2 DD 0
      VAR1     DD 0
      VAR2     DD 0
      VAR3     DD 0
               DD 0
      VAR4     DD 0
      VAR5     DD 0
      FL       DD 0.5e1
      NFILE    db  62h, 0,  6Fh, 0, 6Fh, 0, 6Bh, 0
               db  2Eh, 0, 6Ah, 0, 70h, 0, 67h , 0, 0, 0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;инициализация библиотеки GDIPLUS
      PUSH  0
      PUSH  OFFSET GDIPLUS1
      PUSH  OFFSET GDIPLUS2
      CALL  GdiplusStartup@12
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
REG_CLASS:
;заполнить структуру окна
;стиль
      MOV  WC.CLSSTYLE,stylcl
;процедура обработки сообщений
      MOV  WC.CLSLPFNWNDPROC,OFFSET WNDPROC
      MOV  WC.CLSCBCLSEXTRA,0
      MOV  WC.CLSCBWNDEXTRA,0
      MOV  EAX,HINST
      MOV  WC.CLSHINSTANCE,EAX
;----------пиктограмма окна
      PUSH  IDI_APPLICATION
      PUSH  0
      CALL  LoadIconA@8
      MOV  [WC.CLSHICON], EAX
;----------курсор окна
      PUSH  IDC_CROSS
      PUSH  0
      CALL  LoadCursorA@8
      MOV   WC.CLSHCURSOR, EAX
;----------
      PUSH  RGBW ;цвет кисти
      CALL  CreateSolidBrush@4 ;создать кисть
```

```
      MOV   WC.CLSHBRBACKGROUND,EAX
      MOV   DWORD PTR WC.MENNAME,0
      MOV   DWORD PTR WC.CLSNAME,OFFSET NAM
      PUSH  OFFSET WC
      CALL  RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH  0
      PUSH  HINST
      PUSH  0
      PUSH  0
      PUSH  DY0      ; DY0 - высота окна
      PUSH  DX0      ; DX0 - ширина окна
      PUSH  100      ; координата Y
      PUSH  100      ; координата X
      PUSH  WS_OVERLAPPEDWINDOW
      PUSH  OFFSET TITLENAME  ; имя окна
      PUSH  OFFSET NAM        ; имя класса
      PUSH  0
      CALL  CreateWindowExA@48
;проверка на ошибку
      CMP  EAX,0
      JZ   _ERR
      MOV  NEWHWND, EAX      ; дескриптор окна
;-----------------------------------
      PUSH  SW_SHOWNORMAL
      PUSH  NEWHWND
      CALL  ShowWindow@8     ; показать созданное окно
;-----------------------------------
      PUSH  NEWHWND
      CALL  UpdateWindow@4   ; перерисовать видимую часть окна
;цикл обработки сообщений
MSG_LOOP:
      PUSH  0
      PUSH  0
      PUSH  0
      PUSH  OFFSET MSG
      CALL  GetMessageA@16
      CMP   AX, 0
      JE    END_LOOP
      PUSH  OFFSET MSG
      CALL  TranslateMessage@4
      PUSH  OFFSET MSG
      CALL  DispatchMessageA@4
      JMP   MSG_LOOP
END_LOOP:
;закрыть GDI+
```

```
      PUSH  GDIPLUS2
      CALL  GdiplusShutdown@4
;выход из программы (закрыть процесс)
      PUSH  MSG.MSWPARAM
      CALL  ExitProcess@4
_ERR:
      JMP   END_LOOP
;-----------------------------------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
      CMP  DWORD PTR [EBP+0CH],WM_DESTROY
      JE   WMDESTROY
      CMP  DWORD PTR [EBP+0CH],WM_CREATE
      JE   WMCREATE
      CMP  DWORD PTR [EBP+0CH],WM_PAINT
      JE   WMPAINT
      CMP  DWORD PTR [EBP+0CH],WM_LBUTTONDOWN
      JE   WMLBUTTON
      JMP  DEFWNDPROC
WMLBUTTON:
      CALL GDIPLUSPAINT
      MOV  EAX, 0
      JMP  FINISH
WMPAINT:
      MOV  EAX, 0
      JMP  FINISH
WMCREATE:
      PUSH DWORD PTR [EBP+08H]
      CALL GetDC@4
      MOV  HDC,EAX
      MOV  EAX, 0
      JMP  FINISH
DEFWNDPROC:
      PUSH  DWORD PTR [EBP+14H]
      PUSH  DWORD PTR [EBP+10H]
      PUSH  DWORD PTR [EBP+0CH]
```

```
      PUSH  DWORD PTR [EBP+08H]
      CALL  DefWindowProcA@16
      JMP   FINISH
WMDESTROY:
;выход
      PUSH  HDC
      PUSH  DWORD PTR [EBP+08H]
      CALL  ReleaseDC@8
      PUSH  0
      CALL  PostQuitMessage@4 ;WM_QUIT
      MOV   EAX, 0
FINISH:
      POP  EDI
      POP  ESI
      POP  EBX
      POP  EBP
      RET  16
WNDPROC  ENDP
;процедура вывода графики GDI+
GDIPLUSPAINT PROC
;создать объект GDI+ для вывода графической информации
      PUSH  OFFSET VAR1
      PUSH  HDC
      CALL  GdipCreateFromHDC@8
;создать перо
      MOV   ECX, FL     ; толщина пера в формате float
      PUSH  OFFSET VAR2
      PUSH  0
      PUSH  ECX
      push  0CC000099h  ; цвет пера
      CALL  GdipCreatePen1@16
;рисовать линию
      PUSH  100
      PUSH  200
      PUSH  50
      PUSH  50
      PUSH  VAR2  ; дескриптор пера
      PUSH  VAR1  ; идентификатор объекта GDI+
      CALL  GdipDrawLineI@24
;создать кисть
      PUSH  OFFSET VAR3
      PUSH  0FF0000FFH
      CALL  GdipCreateSolidFill@8
;нарисовать закрашенную окружность
      PUSH  200
      PUSH  200
```

```
      PUSH  100
      PUSH  100
      PUSH  VAR3  ; дескриптор кисти
      PUSH  VAR1  ; идентификатор объекта GDI+
      CALL  GdipFillEllipseI@24
;загрузить изображение из файла
      PUSH  OFFSET VAR4
      PUSH  OFFSET NFILE  ; имя файла в формате Unicode
      CALL  GdipLoadImageFromFile@8
;отобразить
      PUSH  100
      PUSH  400
      PUSH  VAR4   ; идентификатор загруженного изображения
      PUSH  VAR1   ; идентификатор объекта GDI+
      CALL  GdipDrawImageI@16
;удалить изображение из памяти
      PUSH  VAR4
      CALL  GdipDisposeImage@4
;уничтожить кисть
      PUSH  VAR3
      CALL  GdipDeleteBrush@4
;уничтожить перо
      PUSH VAR2
      CALL GdipDeletePen@4
;уничтожить объект GDI+
      PUSH  VAR1
      CALL  GdipDeleteGraphics@4
      RET
GDIPLUSPAINT ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.2.1:

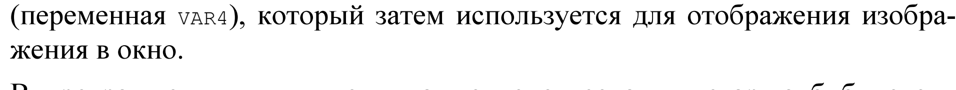
```
ml /c /coff  graph2.asm
link /subsystem:windows graph2.obj
```

Приведу комментарий к программе из листинга 2.2.1.

- Перед тем как использовать GDI+, необходимо инициализировать эту библиотеку. Для этого используется функция `GdiplusStartup`. Функция имеет три параметра. Первый параметр — адрес двойного слова, куда будет помещен некоторый номер (маркер), который затем следует использовать при закрытии библиотеки. Вторым параметром является указатель на 32-байтовую структуру. Первый элемент структуры содержит версию библиотеки GDI+. Туда следует поместить 1. В остальных полях структуры можно, в частности, задать указатель на процедуру, которая получит

управление в случае возникновения ошибки. В простейшем случае следует заполнить эти поля нулями, что я и сделал в программе. Наконец, последний параметр функции можно также положить равным нулю (`NULL`). В конце работы программы следует закрыть библиотеку GDI+ при помощи функции `GdiplusShutdown`. Единственным параметром этой функции является значение маркера, который был получен при выполнении функции `GdiplusStartup`.

- Остальные функции библиотеки GDI+ я сосредоточил в процедуре `GDIPLUSPAINT`, которая выполняется, если щелкнуть по окну левой кнопкой мыши. Обратим внимание, в первую очередь, на функцию `GdipCreateFromHDC`, создающую графический объект библиотеки GDI+, который ассоциируется с конкретным контекстом окна. Описатель этого объекта необходим для изображения конкретных графических объектов в окне. В нашей программе он сохраняется в переменной `VAR1`. В конце процедуры мы удаляем графический объект с помощью функции `GdipDeleteGraphics`. Замечу, что вместо контекста окна можно использовать контекст виртуального окна *(см. главу 2.1)*, а затем, по получению сообщения `WM_PAINT`, следует скопировать это окно на наше окно, обеспечивая тем самым неизменность его содержимого.
- Управление конкретными графическими объектами довольно тривиально и аналогично тому, как это делается в библиотеке GDI.
  - Создать перо — `GdipCreatePen`, рисовать — `GdipDrawLineI`, удалить перо — `GdipDeletePen`. Заметим, что цвет пера (как и кисти) задается 32-битовой величиной (цвет формируется путем наложения 4-х цветов). Обратите внимание, что толщину пера следует задавать в формате `float`. К счастью, MASM 9.0 позволяет задавать такие числовые константы (см. переменную `FL`). При создании пера возвращается ее описатель, который затем используется в других функциях управления пером (см. переменную `VAR2`).
  - Создать кисть — `GdipCreateSolidFill`, нарисовать закрашенный эллипс — `GdipFillEllipseI`, удалить кисть — `GdipDeleteBrush`. Так же как и для пера, при создании кисти возвращается описатель кисти (переменная `VAR3`).
- Отображение графического рисунка, хранящегося в файле, осуществляется по следующей схеме: загрузить изображение из файла в память — `GdipLoadImageFromFile`, отобразить загруженное изображение — `GdipDrawImageI`, освободить память — `GdipDisposeImage`. Обратим внимание, что имя файла (переменная `NFILE`) указано у нас в формате Unicode (имя файла — book.jpg). При загрузке изображения в память возвращается дескриптор


(переменная `VAR4`), который затем используется для отображения изображения в окно.

- В программе мы вынуждены также использовать и старую библиотеку GDI. Это сделано только ради функции `CreateSolidBrush`, которую мы используем для задания цвета окна. Но цвет окна можно задать, просто указав одну из системных кистей. В этом случае отпадает необходимость и в библиотеке GDI.

Библиотека DirectX

Библиотека DirectX представляет собой совокупность технологий, разработанных фирмой Microsoft, для того чтобы превратить Windows в мультимедийную операционную систему. API-функции библиотеки можно разделить на следующие четыре группы:

- DirectDraw — компоненты прямого доступа к видеопамяти. Именно с отдельными представителями из этой группой функций мы познакомимся в данной главе;
- DirectSound — прямой доступ к звуковой карте. Последнее расширение — DirectSound3D. Позволяет позиционировать звук в пространстве, чтобы создавалось впечатление движения объекта;
- DirectPlay — доступ к сетевым ресурсам. Предполагается для использования сетевых игр и мультимедиа;
- DirectInput — поддержка различных устройств ввода;
- Direct3D — компонент для рисования трехмерной графики. Начиная с 8-й версии DirectX, были объединены библиотеки DirectDraw и Direct3D. Теперь эта библиотека называется DirectGraphics;
- DirectMusic — поддержка технологии MIDI.

Остановимся подробнее на библиотеке DirectDraw. Основная задача, которую выполняют функции данной библиотеки, — это прямой доступ к видеопамяти. Основой технологии является копирование из видеопамяти в видеопамять, максимально используя возможности видеоадаптера (включая спрайтовую графику, *z*-буфер и др.) и освобождая центральный процессор. Для ускорения мультимедийной обработки библиотека DirectX широко использует команды MMX *(см. приложение 2)*.

Технология DirectX тесно увязана с основными графическими технологиями Windows GDI и GDI+. В примере, который представлен далее (см. листинг 2.2.2), мы увидим, что можно использовать совместно функции DirectX и GDI на основе единого понятия "контекст".

Замечание

К середине 90-х годов прошлого века и моменту появления Windows 95 большинство игр делалось на основе операционной системы MS-DOS. При этом фирмы-разработчики создавали собственные драйверы для различных аппаратных платформ. Наверное, многие в то время встречались с ситуацией, когда программа не распознавала вашу аппаратуру и отказывалась работать. Вместо создания собственного API Microsoft использовала разработку небольшой компании RenderMorphic. Говорят, что изначально библиотека была создана в рамках некоторого студенческого задания автором, который, в конечном итоге, провалился на экзамене. Тем не менее, Microsoft интегрировала эту библиотеку в свой Game SDK. Корпорация подавала это как идеальное решение для программирования игр. Но прошло несколько лет, прежде чем DirectX стала широко использоваться разработчиками мультимедийных приложений.

Основным понятием технологии DirectX является *поверхность* (surface). Поверхность может содержать различные графические объекты, которые могут быть отображены на экране. Поверхности могут располагаться как в видеопамяти, так и в обычной памяти. Путем специальной операции можно сделать ту или иную поверхность видимой. Если все поверхности располагаются в видеопамяти, то такое переключение происходит очень быстро. Таким образом, возникает эффект плавного изменения изображения, что и требуется для мультимедийных программ. Замечу, что библиотека DirectX автоматически определяет объем видеопамяти и в случае ее нехватки создает поверхность в обычной памяти.

Технология DirectX основывается на модели COM (Component Object Model, модель составных объектов). Однако нам не понадобится изучать эту модель[2]. Достаточно запомнить следующее. После создания объекта DirectX вы получаете указатель (адрес) области, где хранятся адреса функций этого объекта. Зная список и порядок расположения этих функции, мы может использовать их в наших целях.

Для работы с DirectDraw нам понадобится только библиотека импорта ddraw.lib, которую вы можете найти в пакете MASM32 либо создать на основе динамической библиотеки ddraw.ddl с помощью утилиты implib.exe, которая поставляется с некоторыми продуктами Borland. Динамическая же библиотека ddraw.dll, которая непосредственно содержит API-функции DirectDraw, располагается в вашей системе, если, конечно у вас установлена библиотека DirectX.

В листинге 2.2.2 представлена программа, демонстрирующая простейшие возможности DirectX: создается поверхность, в которую помещают битовую картинку. Замечу при этом, что для загрузки картинки в память и копирова-

[2] Всех интересующихся отсылаю к замечательной книге: Д. Роджерсон. Основы COM. — М.: Microsoft "Русская редакция", 1997.

ния ее на поверхность используются обычные API-функции. В результате копирования битовый образ будет отображен на экран (рис. 2.2.2).

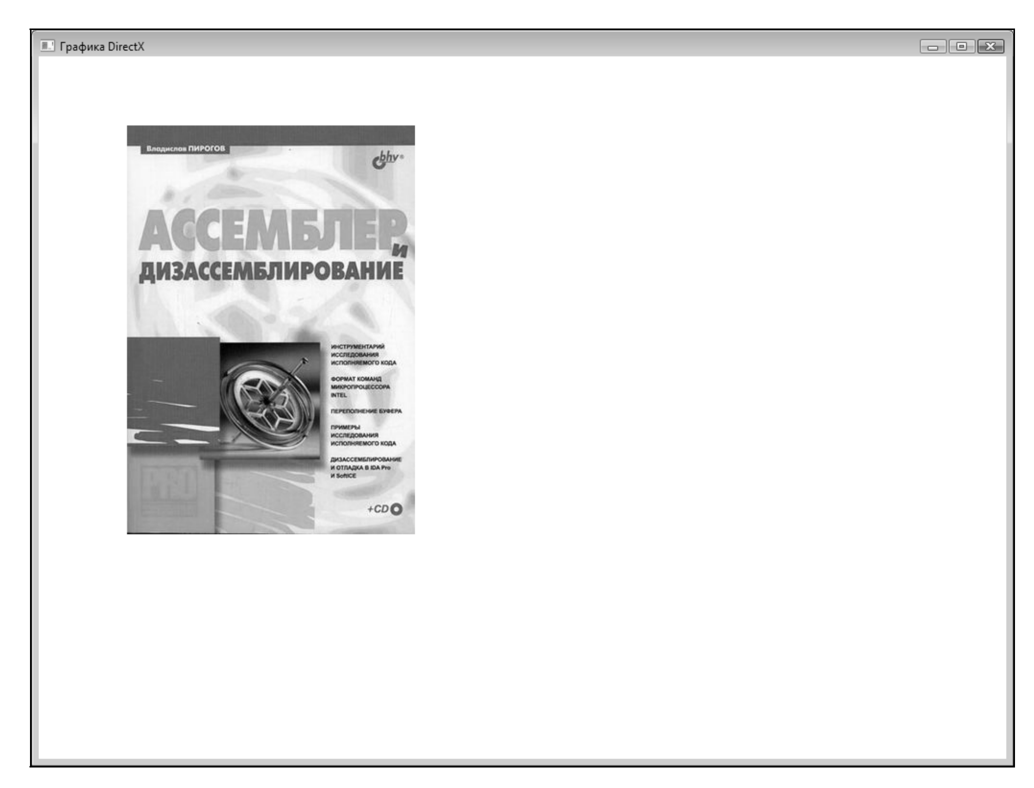


Рис. 2.2.2. Использование библиотеки DirectX для отображения графических образов

Листинг 2.2.2. Пример использования библиотеки DirectX

```
;файл graph2.inc
;константы
;сообщение приходит при закрытии окна
WM_DESTROY     equ 2
;сообщение приходит при создании окна
WM_CREATE      equ 1
;сообщение приходит при перерисовке окна
WM_PAINT       equ 0Fh
;сообщение при щелчке левой кнопкой мыши в области окна
WM_LBUTTONDOWN equ 201h
;свойства окна
CS_VREDRAW     equ 1h
CS_HREDRAW     equ 2h
```

```
CS_GLOBALCLASS equ 4000h
WS_OVERLAPPEDWINDOW equ 000CF0000H
stylcl  equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
DX0            equ 600
DY0            equ 400
;цвета
RED            equ 255
GREEN          equ 255
BLUE           equ 255
RGBW           equ ((RED or (GREEN shl 8)) or (BLUE shl 16))
;идентификатор стандартной пиктограммы
IDI_APPLICATION equ 32512
;идентификатор курсора
IDC_CROSS      equ 32515
;режим показа окна - нормальный
SW_SHOWNORMAL  equ 1
;прототипы внешних процедур
EXTERN  DeleteObject@4:NEAR
EXTERN  DeleteDC@4:NEAR
EXTERN  BitBlt@36:NEAR
EXTERN  SelectObject@8:NEAR
EXTERN  CreateCompatibleDC@4:NEAR
EXTERN  DirectDrawCreate@12:NEAR
EXTERN  GetDC@4:NEAR
EXTERN  ReleaseDC@8:NEAR
EXTERN  MessageBoxA@16:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  CreateSolidBrush@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
EXTERN  LoadImageA@24:NEAR
EXTERN  GetObjectA@12:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?   ; идентификатор окна, получающего сообщение
```

```
      MSMESSAGE  DD ?   ; идентификатор сообщения
      MSWPARAM   DD ?   ; дополнительная информация о сообщении
      MSLPARAM   DD ?   ; дополнительная информация о сообщении
      MSTIME     DD ?   ; время посылки сообщения
      MSPT       DD ?   ; положение курсора во время
                        ; посылки сообщения
MSGSTRUCT ENDS
;---------
WNDCLASS STRUC
      CLSSTYLE        DD ?   ; стиль окна
      CLSLPFNWNDPROC DD ?    ; указатель на процедуру окна
      CLSCBCLSEXTRA   DD ?   ; информация о дополнительных байтах
                             ; для данной структуры
      CLSCBWNDEXTRA   DD ?   ; информация о дополнительных байтах для окна
      CLSHINSTANCE    DD ?   ; дескриптор приложения
      CLSHICON        DD ?   ; идентификатор пиктограммы окна
      CLSHCURSOR      DD ?   ; идентификатор курсора окна
      CLSHBRBACKGROUND DD ? ; идентификатор кисти окна
      MENNAME         DD ?   ; имя-идентификатор меню
      CLSNAME         DD ?   ; специфицирует имя класса окон
WNDCLASS ENDS
;---------------
BITMAP STRUCT
        bmType        DWORD ?
        bmWidth       DWORD ?
        bmHeight      DWORD ?
        bmWidthBytes  DWORD ?
        bmPlanes      WORD  ?
        bmBitsPixel   WORD  ?
        bmBits        DWORD ?
BITMAP ENDS
;directx.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;-------------------------------------------------
include graph3.inc
;подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\gdi32.lib
includelib c:\masm32\lib\ddraw.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      HBM1    DD 0
```

```
      PR      DD 0
      NEWHWND DWORD 0
      MSG     MSGSTRUCT <?>
      WC      WNDCLASS  <?>
      HINST   DD 0
      HDC     DD ?
      HDC1    DD ?
      LP      DD ?
      PDD     DD ?
      HBM     DD ?
      HIM     DD ?
      BM      BITMAP <0>
      DDS     DB 108 DUP(0)  ; структура, используемая
                             ; в функции CreateSurface
      TEXT1   DB '  ', 0
      TEXT2   DB 'Сообщение', 0
      FNAME   DB 'bmp1.bmp',0
      TITLENAME DB 'Графика DirectX',0
      NAM     DB 'CLASS32',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;создать объект DirectX (инициализация)
      PUSH  0
      PUSH  OFFSET LP
      PUSH  0
      CALL  DirectDrawCreate@12
      TEST  EAX,EAX
      JNZ   _ERR
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
REG_CLASS:
;заполнить структуру окна
;стиль
      MOV  WC.CLSSTYLE,stylcl
;процедура обработки сообщений
      MOV  WC.CLSLPFNWNDPROC,OFFSET WNDPROC
      MOV  WC.CLSCBCLSEXTRA,0
      MOV  WC.CLSCBWNDEXTRA,0
      MOV  EAX,HINST
      MOV  WC.CLSHINSTANCE,EAX
;----------пиктограмма окна
      PUSH  IDI_APPLICATION
```

```
      PUSH  0
      CALL  LoadIconA@8
      MOV  [WC.CLSHICON], EAX
;----------курсор окна
      PUSH  IDC_CROSS
      PUSH  0
      CALL  LoadCursorA@8
      MOV   WC.CLSHCURSOR, EAX
;----------
      PUSH  RGBW                ; цвет кисти
      CALL  CreateSolidBrush@4  ; создать кисть
      MOV   WC.CLSHBRBACKGROUND,EAX
      MOV   DWORD PTR WC.MENNAME,0
      MOV   DWORD PTR WC.CLSNAME,OFFSET NAM
      PUSH  OFFSET WC
      CALL  RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH  0
      PUSH  HINST
      PUSH  0
      PUSH  0
      PUSH  DY0      ; DY0 - высота окна
      PUSH  DX0      ; DX0 - ширина окна
      PUSH  100      ; координата Y
      PUSH  100      ; координата X
      PUSH  WS_OVERLAPPEDWINDOW
      PUSH  OFFSET TITLENAME ; имя окна
      PUSH  OFFSET NAM       ; имя класса
      PUSH  0
      CALL  CreateWindowExA@48
;проверка на ошибку
      CMP  EAX,0
      JZ   _ERR
      MOV  NEWHWND, EAX    ; дескриптор окна
;определить уровень доступа к объекту DirectX
      MOV  EAX,LP
      MOV  ECX,[EAX]
;11H - полный доступ (10H) и доступ ко всему экрану (1H)
      PUSH 11H
      PUSH NEWHWND         ; контекст окна
      PUSH EAX
      MOV  EDX,[ECX+50H]   ; функция SetCooperativeLevel
      CALL EDX
;установить видеорежим: 1024*768*32
      MOV  EAX,LP
      MOV  ECX,[EAX]
```

```
      PUSH 32
      PUSH 768
      PUSH 1024
      PUSH EAX
      MOV  EDX,[ECX+54H]   ; функция SetDisplayMode
      CALL EDX
;----------------------------------
      PUSH  SW_SHOWNORMAL
      PUSH  NEWHWND
      CALL  ShowWindow@8   ; показать созданное окно
;----------------------------------
      PUSH  NEWHWND
      CALL  UpdateWindow@4 ; перерисовать видимую часть окна
;цикл обработки сообщений
MSG_LOOP:
      PUSH  0
      PUSH  0
      PUSH  0
      PUSH  OFFSET MSG
      CALL  GetMessageA@16
      CMP   AX, 0
      JE    END_LOOP
      PUSH  OFFSET MSG
      CALL  TranslateMessage@4
      PUSH  OFFSET MSG
      CALL  DispatchMessageA@4
      JMP   MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
      PUSH  MSG.MSWPARAM
      CALL  ExitProcess@4
_ERR:
      JMP  END_LOOP
;----------------------------------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
```

```
      PUSH EDI
      CMP  DWORD PTR [EBP+0CH],WM_DESTROY
      JE   WMDESTROY
      CMP  DWORD PTR [EBP+0CH],WM_CREATE
      JE   WMCREATE
      CMP  DWORD PTR [EBP+0CH],WM_PAINT
      JE   WMPAINT
      CMP  DWORD PTR [EBP+0CH],WM_LBUTTONDOWN
      JE   WMLBUTTON
      JMP  DEFWNDPROC
WMLBUTTON:
      CMP  PR,0
      JNZ  DEFWNDPROC
      MOV  PR,1
;вызов функции создания и отображения поверхности
      CALL DIRECTX
      MOV  EAX, 0
      JMP  FINISH
WMPAINT:
      MOV EAX, 0
      JMP FINISH
WMCREATE:
      PUSH DWORD PTR [EBP+08H]
      CALL GetDC@4
      MOV  HDC,EAX
      MOV  EAX, 0
      JMP  FINISH
DEFWNDPROC:
      PUSH  DWORD PTR [EBP+14H]
      PUSH  DWORD PTR [EBP+10H]
      PUSH  DWORD PTR [EBP+0CH]
      PUSH  DWORD PTR [EBP+08H]
      CALL  DefWindowProcA@16
      JMP   FINISH
WMDESTROY:
;выход
      PUSH  HDC
      PUSH  DWORD PTR [EBP+08H]
      CALL  ReleaseDC@8
      PUSH  0
      CALL  PostQuitMessage@4 ;WM_QUIT
      MOV   EAX, 0
FINISH:
      POP EDI
      POP ESI
      POP EBX
```

```
      POP EBP
      RET 16
WNDPROC  ENDP
;процедура вывода графики
DIRECTX PROC
;заполнить некоторые поля структуры DDS
      LEA  EBX,DDS
;размер структуры
      MOV  DWORD PTR [EBX],108
;поле флагов DDSD_CAPS | DDSD_BACKBUFFERCOUNT
      MOV  DWORD PTR [EBX+4],21H
;поле ddsCaps DDSCAPS_PRIMARYSURFACE | DDSCAPS_FLIP | DDSCAPS_COMPLEX
      MOV  DWORD PTR [EBX+104],218H
;поле dwBackBufferCount
      MOV  DWORD PTR [EBX+20],1
;запуск lpDD->CreateSurface(&ddsd, &pdds, NULL)
      MOV  EAX,LP
      MOV  ECX,[EAX]
      MOV  EDX,[ECX+18H]
      PUSH 0
      PUSH OFFSET PDD
      PUSH OFFSET DDS
      PUSH EAX
      CALL EDX
      TEST EAX,EAX
      JZ   _NO_ERR1
;в случае ошибки запускаем lpDD->Release();
_ERR2:
      MOV  EAX,LP
      MOV  ECX,[EAX]
      MOV  EDX,[ECX+8H]
      PUSH EAX
      CALL EDX
      JMP _EXIT
_NO_ERR1:
;загрузить битовую картинку обычной API-функцией LoadImage из GDI-набора
      PUSH 2010h ;LR_LOADFROMFILE|LR_CREATEDIBSECTION
      PUSH 0
      PUSH 0
      PUSH 0 ;IMAGE_BITMAP
      PUSH OFFSET FNAME
      PUSH 0
      CALL LoadImageA@24
      TEST EAX,EAX
      JZ   _ERR2
      MOV  HBM,EAX
```

```
;получить размер
      PUSH OFFSET BM
      PUSH 24 ;размер структуры BM
      PUSH HBM
      CALL GetObjectA@12
;создаем совместимый контекст устройства
      PUSH 0
      CALL CreateCompatibleDC@4
      MOV  HIM,EAX
;выбрать в контексте
      PUSH HBM
      PUSH EAX
      CALL SelectObject@8
      MOV  HBM1,EAX
;получить контекст поверхности
      MOV  EAX,PDD
      MOV  EDX,[EAX]
      MOV  ECX,[EDX+44H]
      PUSH OFFSET HDC1
      PUSH EAX
      CALL ECX
      TEST EAX,EAX
      JNZ  _ERR2
;копируем загруженное изображение на созданную ранее поверхность
      PUSH 0CC0020h ;SRCCOPY
      PUSH 0
      PUSH 0
      PUSH HIM
      PUSH BM.bmHeight
      PUSH BM.bmWidth
      PUSH 100
      PUSH 100
      PUSH HDC1
      CALL BitBlt@36
      TEST EAX,EAX
      JZ   _ERR2
;вызываем PDD->ReleaseDC
      MOV  EAX,PDD
      MOV  EDX,[EAX]
      MOV  ECX,[EDX+68H]
      PUSH HDC1
      PUSH EAX
      CALL ECX
;выбираем объект в старом контексте
      PUSH HBM1
      PUSH HIM
```

```
      CALL SelectObject@8
;удалить контекст
      PUSH HIM
      CALL DeleteDC@4
;удалить картинку из памяти
      PUSH HBM
      CALL DeleteObject@4
_EXIT:
      RET
DIRECTX ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.2.2:

```
ml /c /coff  graph3.asm
link /subsystem:windows graph3.obj
```

Остановимся подробнее на программе из листинга 2.2.2.

- Работа с DirectDraw начинается с создания основного объекта. Для этого используется API-функция `DirectDrawCreate`. После успешного выполнения функции мы получаем указатель (см. переменную `LP`) на область памяти, где хранятся адреса функций объекта. Вызов функции объекта осуществляется по следующей схеме:

```
;получить адрес
MOV EAX,LP
...
;получить адрес области адресов
MOV ECX,[EAX]
...
; получить адрес функции объекта
MOV  EDX,[ECX+N*4] ;N — номер функции
...
;вызов функции
CALL EDX
```

- Функции `SetCooperativeLevel` и `SetDisplayMode` мы используем для установки начального режима с графикой DirectDraw (см. комментарий к тексту программы).
- Процедура DirectX выполняется, если щелкнуть по окну правой кнопкой мыши. Она предназначена для загрузки в память картинки в формате BMP и копирования ее на поверхность. Тем самым изображение сразу появится на экране.
- Обратим внимание на функцию `CreateSurface`, с помощью которой создается поверхность. Это тоже объект, и доступ к его функциям осуществля-

ется по схеме, которую мы уже разобрали выше. При создании объекта мы использовали структуру `ddsd`. Мы не описываем, а только резервируем данную структуру, ввиду ее громоздкости, а всех интересующихся отсылаем к фундаментальному справочнику MSDN, фирмы Microsoft, где эта структура описана. Мы используем всего три поля данной структуры. Особо обратите внимание на поле `ddsCaps`, значение которого обеспечит нам появление содержимого поверхности (поверхность не теневая) на экране.

- Замечу, что далее мы почти не используем "родные" функции DirectX. Загрузка картинки с диска, определение ее размера, создание контекста — все выполняется стандартными функциями GDI, с которыми мы уже знакомы. Только для получения контекста поверхности мы использовали API-функцию DirectX — `GetDC`, являющуюся функцией объекта "поверхность".
- Наконец, копирование образа на поверхность осуществляется также известной нам API-функцией `BitBlt`. Впрочем, это совсем даже не обязательно, т. к. в арсенале DirectX существует функция `BltFast`, которая выполняет такую операцию гораздо быстрее.
- Обратим внимание на структуру `BM`, которая используется для получения размеров загруженной картинки (см. вызов `GetObject`). Полученные размеры (`BM.bmHeight` и `BM.bmWidth`) далее мы используем при копировании битового образа функцией `BitBlt`.

Программируем на OpenGL

Стандарт OpenGL (Open Graphic Library, открытая графическая библиотека) был принят в 1992 году. Основой данного стандарта послужила библиотека IRIS GL[3], разработанная фирмой Silicon Graphic Inc. (SGI). В дальнейшем стандарт OpenGL подвергался неоднократной переработке комиссией, состоящих из нескольких ведущих производителей программного[4] обеспечения, в том числе и фирмой Microsoft, которая поставляет одну из реализаций OpenGL вместе со своими операционными системами[5]. Особенностью стан-

[3] Когда-то эта библиотека использовалась на мощных графических станциях Silicon Graphics.

[4] В состав комиссии входили такие известные фирмы, как Sun Microsystems, Hewlett Packard Corp., IBM Corp., Intel Corp., Digital Equipment Corporation и др.

[5] Лишний раз приходится поаплодировать политике, проводимой фирмой Microsoft. Ведь поставка OpenGL вместе с операционной системой значительно облегчает как разработку приложений на базе OpenGL, так и их распространение.

дарта OpenGL является полная совместимость с ранее выпущенными версиями. В настоящее время между библиотеками DirectX и OpenGL сложилось разделение труда. Если библиотека DirectX преимущественно ется для программирования игровых программ, то библиотека OpenGL в большей степени — в научной и инженерной графике.

Библиотека OpenGL поддерживает следующие возможности:

- набор базовых примитивов;
- видовые и координатные преобразования;
- удаление невидимых линий и поверхностей;
- наложение текстуры и применение освещения;
- специальные эффекты: туман, изменение прозрачности, сопряжение цветов (плавный переход между цветами).

К сожалению, я продемонстрирую только основы использования OpenGL, необходимые для самостоятельного освоения данной технологии, тем более, что для полноценного использования библиотеки требуются знания в области двумерной и трехмерной геометрий.

Основной интерфейс OpenGL сосредоточен в двух динамических библиотеках, поставляемых вместе с Windows: opengl32.dll и glu32.dll. Соответствующие библиотеки импорта opengl32.lib и glu32.lib можно найти в пакете MASM32. Как и DirectX, библиотека OpenGL (для Windows) не является самодостаточной, а интегрируется в систему API Windows. Далее вы увидите, что часть функций, используемых специально для инициализации OpenGL, являются просто частью интерфейса API Windows. В отличие от DirectX имеются реализации OpenGL для других операционных систем, например UNIX, а также консольных приложений. Существуют реализации библиотеки и для компьютеров фирмы Apple. Кроме этого, библиотека OpenGL может использоваться в сетевом варианте: прикладная программа-клиент на одном компьютере и сервер, исполняющий команды OpenGL, — на другом.

В основе функционирования библиотеки OpenGL лежит понятие графического примитива. В примере из листинга 2.2.3 мы выводим простейший графический примитив — треугольник. Графические примитивы задаются своими вершинами (массивом вершин). Основные функции OpenGL строятся по следующей схеме:

```
glCommand_name[1 2 3 4][b s i f d ub us ui][v] (arg1, arg2, ..., argn)
```

Здесь:

- `gl` — обязательный префикс;
- *`Command_name`* — имя команды;

- `[1 2 3 4]` — количество аргументов команды;
- `[b s i f d ub us ui]` — тип аргумента: `ui` — unsigned integer, `f` — float, `b` — тип byte и т. д.;
- `v` — наличие этого символа означает, что в качестве одного из параметров функции используется указатель на массив значений.

В листинге 2.2.3 представлена программа, выводящая простой примитив — треугольник — в окно приложения (рис. 2.2.3). Приложение построено таким образом, чтобы вы могли, взяв его за основу, программировать более сложные графические образы, в том числе с элементами анимации. Как и ранее, вы можете поработать с приложением, которое имеется на прилагаемом к книге компакт-диске.

Листинг 2.2.3. Пример использования библиотеки OpenGL

```
;файл graph2.inc (opengl)
;константы
PFD_DRAW_TO_WINDOW    equ 00000004h
PFD_SUPPORT_OPENGL    equ 00000020h
PFD_DOUBLEBUFFER      equ 00000001h
PFD      equ   PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL  or PFD_DOUBLEBUFFER
PFD_MAIN_PLANE        equ 0
PFD_TYPE_RGBA         equ 0
GL_TRIANGLES          equ  00004h
GL_COLOR_BUFFER_BIT   equ  000004000h
GL_MODELVIEW_MATRIX   equ  00BA6h
GL_VERTEX_ARRAY       equ  08074h
GL_FLOAT              equ  01406h
GL_UNSIGNED_SHORT     equ  01403h
GL_COLOR_MATERIAL     equ  0B57h
;сообщение приходит при установке размеров окна
WM_SIZE               equ 5h
;сообщение приходит при закрытии окна
WM_DESTROY            equ 2
;сообщение приходит при создании окна
WM_CREATE             equ 1
;сообщение приходит при перерисовке окна
WM_PAINT              equ 0Fh
;сообщение при щелчке левой кнопкой мыши в области окна
WM_LBUTTONDOWN        equ 201h
;свойства окна
WS_VISIBLE            equ 090000000h
CS_VREDRAW            equ 1h
CS_HREDRAW            equ 2h
```

```
CS_GLOBALCLASS        equ 4000h
WS_OVERLAPPEDWINDOW   equ 000CF0000H
WS_CLIPSIBLINGS       equ 4000000h
WS_CLIPCHILDREN       equ 2000000h
stylcl  equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
DX0                   equ 600
DY0                   equ 400
PM_NOREMOVE           equ 0h
;цвета
RED                   equ 255
GREEN                 equ 255
BLUE                  equ 255
RGBW                  equ ((RED or (GREEN shl 8)) or (BLUE shl 16))
;идентификатор стандартной пиктограммы
IDI_APPLICATION       equ 32512
;идентификатор курсора
IDC_CROSS             equ 32515
;режим показа окна - нормальный
SW_SHOWNORMAL         equ 1
;прототипы внешних процедур
EXTERN  glDrawArrays@12:NEAR
EXTERN  BeginPaint@8:NEAR
EXTERN  EndPaint@8:NEAR
EXTERN  PeekMessageA@20:NEAR
EXTERN  glVertexPointer@16:NEAR
EXTERN  glEnableClientState@4:NEAR
EXTERN  wglDeleteContext@4:NEAR
EXTERN  glClearColor@16:NEAR
EXTERN  glShadeModel@4:NEAR
EXTERN  glLoadIdentity@0:NEAR
EXTERN  glMatrixMode@4:NEAR
EXTERN  glEnable@4:NEAR
EXTERN  glClear@4:NEAR
EXTERN  SwapBuffers@4:NEAR
EXTERN  glColor3ub@12:NEAR
EXTERN  wglCreateContext@4:NEAR
EXTERN  wglMakeCurrent@8:NEAR
EXTERN  SetPixelFormat@12:NEAR
EXTERN  ChoosePixelFormat@8:NEAR
EXTERN  GetDC@4:NEAR
EXTERN  ReleaseDC@8:NEAR
EXTERN  MessageBoxA@16:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  CreateSolidBrush@4:NEAR
```

```
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?  ; идентификатор окна, получающего сообщение
      MSMESSAGE  DD ?  ; идентификатор сообщения
      MSWPARAM   DD ?  ; дополнительная информация о сообщении
      MSLPARAM   DD ?  ; дополнительная информация о сообщении
      MSTIME     DD ?  ; время посылки сообщения
      MSPT       DD ?  ; положение курсора во время
;посылки сообщения
MSGSTRUCT ENDS
;---------
WNDCLASS STRUC
      CLSSTYLE      DD ?   ; стиль окна
      CLSLPFNWNDPROC DD ?  ; указатель на процедуру окна
      CLSCBCLSEXTRA DD ?   ; информация о доп. байтах
                           ; для данной структуры
      CLSCBWNDEXTRA DD ?   ; информация о дополнительных байтах для окна
      CLSHINSTANCE  DD ?   ; дескриптор приложения
      CLSHICON      DD ?   ; идентификатор пиктограммы окна
      CLSHCURSOR    DD ?   ; идентификатор курсора окна
      CLSHBRBACKGROUND  DD ?  ; идентификатор кисти окна
      MENNAME       DD ?   ; имя-идентификатор меню
      CLSNAME       DD ?   ; специфицирует имя класса окон
WNDCLASS ENDS
;---------------
PIXELFORMATDESCRIPTOR STRUCT
        nSize           WORD  ?
        nVersion        WORD  ?
        dwFlags         DWORD ?
        iPixelType      BYTE  ?
        cColorBits      BYTE  ?
        cRedBits        BYTE  ?
        cRedShift       BYTE  ?
        cGreenBits      BYTE  ?
        cGreenShift     BYTE  ?
```

```
        cBlueBits        BYTE  ?
        cBlueShift       BYTE  ?
        cAlphaBits       BYTE  ?
        cAlphaShift      BYTE  ?
        cAccumBits       BYTE  ?
        cAccumRedBits    BYTE  ?
        cAccumGreenBits  BYTE  ?
        cAccumBlueBits   BYTE  ?
        cAccumAlphaBits  BYTE  ?
        cDepthBits       BYTE  ?
        cStencilBits     BYTE  ?
        cAuxBuffers      BYTE  ?
        iLayerType       BYTE  ?
        bReserved        BYTE  ?
        dwLayerMask      DWORD ?
        dwVisibleMask    DWORD ?
        dwDamageMask     DWORD ?
PIXELFORMATDESCRIPTOR ENDS
;-----
PAINTSTR STRUC
      Hdc     DD 0
      fErase  DD 0
      left    DD 0
      top     DD 0
      right   DD 0
      bottom  DD 0
      fRes    DD 0
      fIncUp  DD 0
      Reserv  DB 32 dup(0)
PAINTSTR ENDS
;opengl.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;-------------------------------------------------
include graph3.inc
;подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\gdi32.lib
includelib c:\masm32\lib\opengl32.lib
includelib c:\masm32\lib\glu32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      PIXFR   PIXELFORMATDESCRIPTOR <0>
```

```
      PIXFORM DD ?
      NEWHWND DD 0
      MSG     MSGSTRUCT <?>
      WC      WNDCLASS  <?>
      PNT     PAINTSTR  <?>
      HINST   DD 0
      HDC     DD ?
      HDC1    DD ?
      TITLENAME DB 'Графика OpenGL',0
      NAM     DB 'CLASS32',0
      GL1     DD  -0.5f,-0.5f,0.0f, 0.5f,0.0f,0.0f, -0.5f,0.5f,0.0f
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
REG_CLASS:
;заполнить структуру окна
;стиль
      MOV  WC.CLSSTYLE,stylcl
;процедура обработки сообщений
      MOV  WC.CLSLPFNWNDPROC,OFFSET WNDPROC
      MOV  WC.CLSCBCLSEXTRA,0
      MOV  WC.CLSCBWNDEXTRA,0
      MOV  EAX,HINST
      MOV  WC.CLSHINSTANCE,EAX
;пиктограмма окна
      PUSH  IDI_APPLICATION
      PUSH  0
      CALL  LoadIconA@8
      MOV  [WC.CLSHICON], EAX
;курсор окна
      PUSH  IDC_CROSS
      PUSH  0
      CALL  LoadCursorA@8
      MOV   WC.CLSHCURSOR, EAX
;цвет кисти
      PUSH  RGBW
      CALL  CreateSolidBrush@4 ;создать кисть
      MOV   WC.CLSHBRBACKGROUND,EAX
      MOV   DWORD PTR WC.MENNAME,0
      MOV   DWORD PTR WC.CLSNAME,OFFSET NAM
      PUSH  OFFSET WC
      CALL  RegisterClassA@4
```

```
;создать окно зарегистрированного класса
      PUSH  0
      PUSH  HINST
      PUSH  0
      PUSH  0
      PUSH  DY0      ; DY0 - высота окна
      PUSH  DX0      ; DX0 - ширина окна
      PUSH  100      ; координата Y
      PUSH  100      ; координата X
      PUSH  WS_VISIBLE or WS_OVERLAPPEDWINDOW or WS_CLIPSIBLINGS or WS_CLIPCHILDREN
      PUSH  OFFSET TITLENAME ; имя окна
      PUSH  OFFSET NAM       ; имя класса
      PUSH  0
      CALL  CreateWindowExA@48
;проверка на ошибку
      CMP  EAX,0
      JZ   _EXIT1
      MOV  NEWHWND, EAX      ; дескриптор окна
;вывести окно
      PUSH  SW_SHOWNORMAL
      PUSH  NEWHWND
      CALL  ShowWindow@8     ; показать созданное окно
;перерисовать видимую часть окна
      PUSH  NEWHWND
      CALL  UpdateWindow@4   ; перерисовать видимую часть окна
;цикл обработки сообщений
MSG_LOOP:
      PUSH  PM_NOREMOVE
      PUSH  0
      PUSH  0
      PUSH  0
      PUSH  OFFSET MSG
;проверить очередь команд
      CALL  PeekMessageA@20
      CMP   EAX,0
      JZ    NO_MES
      PUSH  0
      PUSH  0
      PUSH  0
      PUSH  OFFSET MSG
;взять сообщение из очереди команд
      CALL  GetMessageA@16
      CMP   AX, 0
      JZ   _EXIT1
      PUSH  OFFSET MSG
      CALL  TranslateMessage@4
```

```
      PUSH  OFFSET MSG
      CALL  DispatchMessageA@4
      JMP   MSG_LOOP
NO_MES:
;инициализировать теневое окно
      CALL INITPAINT
;осуществить вывод изображения
      CALL OPENGL
      JMP  MSG_LOOP
_EXIT1:
;выход из программы (закрыть процесс)
      PUSH  MSG.MSWPARAM
      CALL  ExitProcess@4
;-----------------------------------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
      CMP  DWORD PTR [EBP+0CH],WM_DESTROY
      JE   WMDESTROY
      CMP  DWORD PTR [EBP+0CH],WM_CREATE
      JE   WMCREATE
      CMP  DWORD PTR [EBP+0CH],WM_PAINT
      JE   WMPAINT
      CMP  DWORD PTR [EBP+0CH],WM_LBUTTONDOWN
      JE   WMLBUTTON
      JMP  DEFWNDPROC
WMLBUTTON:
      MOV  EAX, 0
      JMP  FINISH
WMPAINT:
      PUSH OFFSET PNT
      PUSH DWORD PTR [EBP+08H]
      CALL BeginPaint@8
;инициализация OpenGL
      CALL INITGL
      PUSH OFFSET PNT
```

```
      PUSH DWORD PTR [EBP+08H]
      CALL EndPaint@8
      MOV  EAX, 0
      JMP  FINISH
WMCREATE:
      MOV EAX, 0
      JMP FINISH
DEFWNDPROC:
      PUSH  DWORD PTR [EBP+14H]
      PUSH  DWORD PTR [EBP+10H]
      PUSH  DWORD PTR [EBP+0CH]
      PUSH  DWORD PTR [EBP+08H]
      CALL  DefWindowProcA@16
      JMP   FINISH
WMDESTROY:
;выход
;текущий контекст - HDC - контекст окна
      PUSH  0
      PUSH  HDC
      CALL  wglMakeCurrent@8
;удалить контекст HDC1
      PUSH  HDC1
      CALL  wglDeleteContext@4
;освободить контекст окна
      PUSH  HDC
      PUSH  DWORD PTR [EBP+08H]
      CALL  ReleaseDC@8
;послать сообщение выхода
      PUSH  0
      CALL  PostQuitMessage@4 ;WM_QUIT
      MOV   EAX, 0
FINISH:
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      RET 16
WNDPROC  ENDP
;процедура вывода графики OpenGL
OPENGL PROC
;установить цвет рисования (белый)
      PUSH  255
      PUSH  255
      PUSH  255
      CALL  glColor3ub@12
```

```
;установить массив вершин многоугольника
      PUSH  OFFSET GL1
      PUSH  0
      PUSH  GL_FLOAT
      PUSH  3
      CALL  glVertexPointer@16
;вывести треугольник
      PUSH  3
      PUSH  0
      PUSH  GL_TRIANGLES
      CALL  glDrawArrays@12
;скопировать теневое окно на видимое окно
      PUSH  HDC
      CALL  SwapBuffers@4
;здесь можно включить алгоритм преобразования координат
      CALL   ADDSTEP
      RET
OPENGL ENDP
;инициализация (задание) пиксельного формата
INITGL PROC
;получить контекст устройства
      PUSH NEWHWND
      CALL GetDC@4
      MOV  HDC,EAX
;выбор пиксельного формата с задаваемыми параметрами
      MOV   AX,sizeof PIXFR
      MOV   PIXFR.nSize,AX    ; размер структуры
      MOV   PIXFR.nVersion,1
      MOV   PIXFR.dwFlags,PFD
      MOV   PIXFR.iPixelType,PFD_TYPE_RGBA
      MOV   PIXFR.cColorBits,32
      MOV   PIXFR.cStencilBits,32
      MOV   PIXFR.cDepthBits,32
      PUSH  OFFSET PIXFR
      PUSH  HDC
      CALL  ChoosePixelFormat@8
;настраиваем контекст устройства на работу с созданным
;пиксельным форматом
      MOV   PIXFORM,EAX
      PUSH  OFFSET PIXFR
      PUSH  PIXFORM
      PUSH  HDC
      CALL  SetPixelFormat@12
;создаем контекст изображения средствами OpenGL, совместимый с HDC
      PUSH  HDC
```

```
      CALL  wglCreateContext@4
      MOV   HDC1,EAX
;делаем созданный контекст активным и указываем, куда,
;в конечном итоге, будет выводиться графика
      PUSH  HDC1
      PUSH  HDC
      CALL  wglMakeCurrent@8
;включить состояние OpenGL
      PUSH  GL_COLOR_MATERIAL ;разрешить использование цвета
      CALL  glEnable@4
      RET
INITGL ENDP
;процедура инициализации теневого окна
INITPAINT  PROC
;очищаем теневое окно для хранения цветового
;изображения GL_COLOR_BUFFER_BIT
      PUSH  GL_COLOR_BUFFER_BIT
      CALL  glClear@4
;установка матричного режима (объектно-видовая матрица)
      PUSH  GL_MODELVIEW_MATRIX
      CALL  glMatrixMode@4
;заменить текущую матрицу на единичную
      CALL  glLoadIdentity@0
;установить цвет, которым будет заполняться экран
      PUSH  0
      PUSH  0
      PUSH  0
      PUSH  0
      CALL  glClearColor@16
;подключение режима массивов вершин
      PUSH  GL_VERTEX_ARRAY
      CALL  glEnableClientState@4
      RET
INITPAINT  ENDP
;процедура преобразования координат графических фигур
ADDSTEP PROC
      RET
ADDSTEP ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.2.3:

```
ml /c /coff  graph+.asm
link /subsystem:windows graph+.obj
```

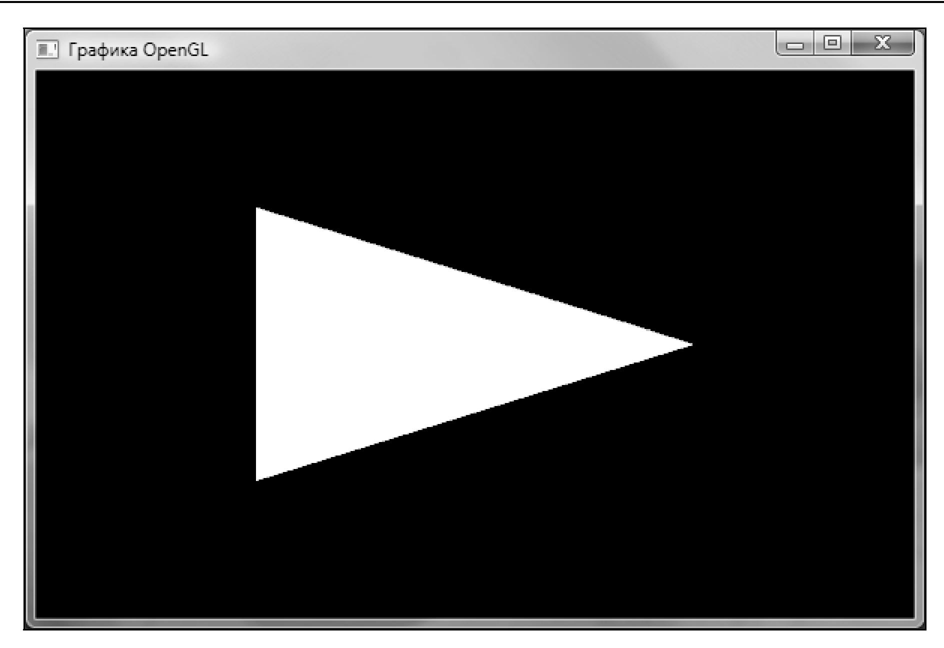
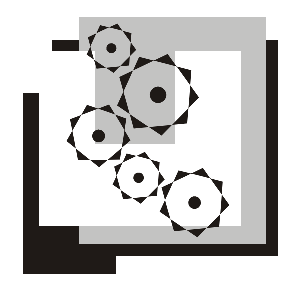


Рис. 2.2.3. Графический объект — треугольник, нарисованный средствами OpenGL

Прокомментирую программу из листинга 2.2.3, остановившись на ключевых моментах.

- Заметим, что при создании окна мы добавили следующие стили: `WS_CLIPSIBLINGS` и `WS_CLIPCHILDREN`. В нашем случае одного окна можно обойтись и без этих стилей. Они вам понадобятся, если вы будете создавать дочерние данному окну окна.
- Обратим внимание на три процедуры: `INITGL`, `INITPAINT`, `OPENGL`. Если кратко, то первая процедура вызывается для инициализации пиксельного формата, вторая — для инициализации теневого окна и третья — для вывода графических примитивов.
- Обратите внимание на то, как построена структура цикла обработки сообщений. Эту структуру мы уже разбирали в *главе 1.2 (см. разд. "Еще о цикле обработки сообщений")*. Теперь мы видим эту структуру в действии. Вывод информации в окно как раз осуществляется в этом цикле. Обратите внимание на процедуру `ADDSTEP`. Эта процедура пуста, но если поместить туда некоторый алгоритм преобразований координат и других атрибутов изображения, то мы получим "живую" картинку, т. е. анимацию. Именно такой прием чаще всего применяется в анимационных программах.

- Обратите внимание, как задается треугольник в программе. Он задается тройками координат каждой вершины (адрес `GL1`). Заметим, есть и третья координата по оси *z*, но в нашем случае плоского изображения она просто игнорируется. Обратим также внимание, что система координат расположена в центре окна, в отличие, например, от GDI. Кроме этого, масштаб нормирован на единицу относительно размеров окна. Координаты, таким образом, должны представляться числами типа `float`.
- Следует, наконец, уделить внимание и функции `SwapBuffers`. Эта функция выводит в окно, представленное контекстом `HDC`, содержимое теневого окна (буфера). Буфер представлен контекстом `HDC1`, который увязан с контекстом `HDC` (функции `wglCreateContext` и `wglMakeCurrent`).



Глава 2.3

Консольные приложения

Консольные программы — что это такое? "О, это для тех, кто любит работать с командной строкой", — скажут некоторые. Консоль — это окно, предназначенное для вывода текстовой информации. Самая известная консольная программа — это файловый менеджер Far.exe. Но дело ведь не только в любви к текстовому режиму. Часто нет необходимости и времени для создания графического интерфейса, а программа должна что-то делать, например, обрабатывать большие объемы данных, а также выводить какую-нибудь текстовую информацию пользователю. И вот тут на помощь приходят консольные приложения. Далее вы увидите, что консольные приложения очень компактны не только в откомпилированном виде, но и в текстовом варианте. Но главное, консольное приложение имеет такие же возможности обращаться к ресурсам Windows посредством API-функций, как и обычное графическое приложение.

Получить консольное приложение довольно просто:

```
ml /c /coff cons1.asm
link /subsystem:console cons1.obj
```

Как и раньше, мы предполагаем, что библиотеки будут указываться при помощи директивы `includelib`. В листинге 2.3.1 представлено простое консольное приложение для MASM32. Для вывода текстовой информации используется API-функция `WriteConsoleA`, параметры которой (слева направо) имеют следующий смысл.

- 1-й параметр — дескриптор буфера вывода консоли, который может быть получен при помощи функции `GetStdHandle`;
- 2-й параметр — указатель на буфер, где находится выводимый текст;
- 3-й параметр — количество выводимых символов;

- 4-й параметр указывает на переменную DWORD, куда будет помещено количество действительно выведенных символов;
- 5-й параметр — резервный параметр, должен быть равен нулю.

Замечу, что буфер, где находится выводимый текст, не обязательно должен заканчиваться нулем, поскольку для данной функции указывается количество выводимых символов. Договоримся только не путать входные/выходные буферы консоли и буферы, которые мы создаем в программе, в том числе и для обмена с буферами консоли.

Листинг 2.3.1. Простое консольное приложение для MASM32

```
;cons1.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
;прототипы внешних процедур
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
;строка в OEM-кодировке
      STR1 DB "Консольное приложение",0
      LENS DD ? ;количество выведенных символов
      RES  DD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить HANDLE вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
;длина строки
      PUSH OFFSET STR1
      CALL LENSTR
;вывести строку
      PUSH OFFSET RES    ; резерв
      PUSH OFFSET LENS   ; выведено символов
```

```
      PUSH EBX          ; длина строки
      PUSH OFFSET STR1  ; адрес строки
      PUSH EAX          ; дескриптор вывода
      CALL WriteConsoleA@20
      PUSH 0
      CALL ExitProcess@4
;строка - [EBP+08H]
;длина в EBX
LENSTR PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EAX
      PUSH EDI
;--------------------
      CLD
      MOV   EDI,DWORD PTR [EBP+08H]
      MOV   EBX,EDI
      MOV   ECX,100   ; ограничить длину строки
      XOR   AL,AL
      REPNE SCASB     ; найти символ 0
      SUB   EDI,EBX   ; длина строки, включая 0
      MOV   EBX,EDI
      DEC   EBX
;--------------------
      POP EDI
      POP EAX
      POP EBP
      RET 4
LENSTR ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.3.1:

```
ml /c /coff cons1.asm
link /subsystem:console cons1.obj
```

Надо сказать, что, поскольку информация выводится в консольном окне, кодировка всех строковых констант должна быть DOS (точнее, кодировкой OEM, *см. главу 1.5*), либо в процессе выполнения программы должно осуществляться динамическое перекодирование.

Прокомментируем теперь приведенную выше программу. При запуске ее из командной строки, например из программы Far, в строку выводится сообщение "Консольное приложение". При запуске программы как Windows-приложения (например, из папки **Мой компьютер**) консольное окно появля-

ется лишь на секунду. В чем тут дело? Дело в том, что консольные приложения могут создавать свою консоль. В этом случае весь ввод/вывод будет производиться в эту консоль. Если же приложение консоль не создает, то здесь может возникнуть двоякая ситуация: либо наследуется консоль, в которой программа была запущена (консоль программы, запустившей приложение), либо Windows создает для приложения свою консоль. При запуске нашей программы не из консоли операционная система Windows создает для запущенного приложения свою консоль, которая закрывается сразу же после окончания работы приложения. В случае запуска из программы Far.exe наше консольное приложение просто наследует консоль far и, соответственно, весь вывод будет осуществляться в эту консоль. После закрытия приложения консоль, разумеется, остается вместе с выведенной туда информацией.

Замечание

Возникает законный вопрос: как операционная система узнает, какое приложение запущено: консольное или графическое? Дело в том, что исполняемые модули имеют заголовок, в котором содержится различная служебная информация, используемая системой. В частности в этом заголовке и содержится информации о типе приложения. Заголовок же формируется во время компиляции, когда мы указываем параметр `CONSOLE` или `WINDOWS`.

Создание консоли

Рассмотрим несколько простых консольных API-функций и их применение.

Работать с чужой (наследуемой) консолью не всегда удобно. А для того чтобы создать свою консоль, используется функция `AllocConsole`. По завершении программы все выделенные консоли автоматически освобождаются. Однако это можно сделать и принудительно, используя функцию `FreeConsole`. Для того чтобы получить дескриптор консоли, предназначена уже знакомая вам функция `GetStdHandle`, аргументом которой может являться следующая из трех констант:

```
STD_INPUT_HANDLE   equ -10 ; для ввода
STD_OUTPUT_HANDLE  equ -11 ; для вывода
STD_ERROR_HANDLE   equ -12 ; для сообщения об ошибке
```

Следует отметить, что один процесс может иметь (наследовать или создавать) только одну консоль, поэтому выполнение в начале программы функции `FreeConsole` обязательно. При запуске программы в "чужой" консоли она наследует эту консоль, поэтому, пока мы не выполним функцию `FreeConsole`, новой консоли не создать — чужую консоль эта функция закрыть не может.

Для чтения из буфера консоли используется функция `ReadConsole`. Значения параметров этой функции (слева направо)[1] следующие:

- 1-й параметр — дескриптор входного буфера;
- 2-й параметр — адрес буфера, куда будет помещена вводимая информация;
- 3-й параметр — длина этого буфера;
- 4-й параметр — количество фактически прочитанных символов;
- 5-й параметр — зарезервировано.

Установить позицию курсора в консоли можно при помощи функции `SetConsoleCursorPosition` со следующими параметрами:

- 1-й параметр — дескриптор входного буфера консоли;
- 2-й параметр — структура `COORD`:

```
COORD STRUC
      X WORD ?
      Y WORD ?
COORD ENDS
```

Хочу лишний раз подчеркнуть, что вторым параметром является *не указатель* на структуру (что обычно бывает), а именно *структура*. На самом деле для ассемблера это просто двойное слово (`DWORD`), у которого младшее слово — координата `X`, а старшее слово — координата `Y`.

Установить цвет выводимых букв можно с помощью функции `SetConsoleTextAttribute`. Первым параметром этой функции является дескриптор выходного буфера консоли, а вторым — цвет букв и фона. Цвет получается путем комбинации (сумма или операция "ИЛИ") двух или более из представленных ниже констант. Причем возможна "смесь" не только цвета и интенсивности, но и цветов (см. программу из листинга 2.3.2).

```
FOREGROUND_BLUE           equ 1h   ; синий цвет букв
FOREGROUND_GREEN          equ 2h   ; зеленый цвет букв
FOREGROUND_RED            equ 4h   ; красный цвет букв
FOREGROUND_INTENSITY      equ 8h   ; повышенная интенсивность
BACKGROUND_BLUE           equ 10h  ; синий свет фона
BACKGROUND_GREEN          equ 20h  ; зеленый цвет фона
BACKGROUND_RED            equ 40h  ; красный цвет фона
BACKGROUND_INTENSITY      equ 80h  ; повышенная интенсивность
```

[1] Вообще, как вы понимаете, для ассемблера практически все параметры имеют тип `DWORD`. По смыслу же они или адреса, или значения. Поэтому проще перечислять их с указанием смысла, чем записывать функцию в С-нотации.

Для определения заголовка окна консоли применяется функция `SetConsoleTitle`, единственным параметром которой является адрес строки с нулем на конце. Здесь имеет смысл оговорить следующее: для вывода строк и в консоль, и в качестве заголовка требуется кодировка OEM. Чтобы покончить с этой проблемой раз и навсегда, посмотрим, как это можно решить средствами Windows.

Существует уже знакомая вам функция `CharToOem` *(см. главу 1.5)*. Первым параметром этой функции является указатель на строку, которую следует перекодировать, а вторым параметром — на строку, в которую нужно поместить результат. Причем поместить результат можно и в строку, которую перекодируем. Вот и все, проблема перекодировки решена. В дальнейшем, в консольных приложениях, мы будем использовать данную функцию без особых оговорок.

Мы рассмотрели несколько консольных функций, всего их около пятидесяти. Нет нужды говорить обо всех этих функциях. О некоторых из них я еще скажу, но читатель, я думаю, по приведенным в книге примерам и обсуждениям сможет сам использовать в своих программах другие консольные функции. Замечу только, что для большинства консольных функций характерно то, что при правильном их завершении возвращается ненулевое значение. В случае ошибки в регистр `EAX` помещается ноль.

Ну что же, пора приступать к разбору следующих примеров, в одном из них программа создает собственную консоль (листинг 2.3.2).

Листинг 2.3.2. Пример создания собственной консоли

```
;cons2.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
STD_INPUT_HANDLE  equ -10
;атрибуты цветов
FOREGROUND_BLUE   equ 1h    ; синий цвет букв
FOREGROUND_GREEN  equ 2h    ; зеленый цвет букв
FOREGROUND_RED    equ 4h    ; красный цвет букв
FOREGROUND_INTENSITY equ 8h ; повышенная интенсивность
BACKGROUND_BLUE   equ 10h   ; синий свет фона
BACKGROUND_GREEN  equ 20h   ; зеленый цвет фона
BACKGROUND_RED    equ 40h   ; красный цвет фона
BACKGROUND_INTENSITY equ 80h ; повышенная интенсивность
COL1 = 2h+8h    ; цвет выводимого текста
```

```
COL2 = 1h+2h+8h ; другой цвет выводимого текста
;прототипы внешних процедур
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  SetConsoleCursorPosition@8:NEAR
EXTERN  SetConsoleTitleA@4:NEAR
EXTERN  FreeConsole@0:NEAR
EXTERN  AllocConsole@0:NEAR
EXTERN  CharToOemA@8:NEAR
EXTERN  SetConsoleCursorPosition@8:NEAR
EXTERN  SetConsoleTextAttribute@8:NEAR
EXTERN  ReadConsoleA@20:NEAR
EXTERN  SetConsoleScreenBufferSize@8:NEAR
EXTERN  ExitProcess@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
COOR  STRUC
      X  WORD ?
      Y  WORD ?
COOR  ENDS
;сегмент данных
_DATA SEGMENT
      HANDL  DWORD ?
      HANDL1 DWORD ?
      STR1   DB "Введите строку:",13,10,0
      STR2   DB "Простой пример работы консоли",0
      BUF    DB 200 dup(?)
      LENS   DWORD ?   ; количество выведенных символов
      CRD    COOR <?>
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;перекодируем строки
        PUSH OFFSET STR1
        PUSH OFFSET STR1
        CALL CharToOemA@8
        PUSH OFFSET STR2
        PUSH OFFSET STR2
        CALL CharToOemA@8
;образовать консоль
;вначале освободить уже существующую
        CALL FreeConsole@0
        CALL AllocConsole@0
```

```
;получить HANDL1 ввода
        PUSH STD_INPUT_HANDLE
        CALL GetStdHandle@4
        MOV  HANDL1,EAX
;получить HANDL вывода
        PUSH STD_OUTPUT_HANDLE
        CALL GetStdHandle@4
        MOV  HANDL,EAX
;установить новый размер окна консоли
        MOV  CRD.X,100
        MOV  CRD.Y,25
        PUSH CRD
        PUSH EAX
        CALL SetConsoleScreenBufferSize@8
;задать заголовок окна консоли
        PUSH OFFSET STR2
        CALL SetConsoleTitleA@4
;установить позицию курсора
        MOV  CRD.X,0
        MOV  CRD.Y,10
        PUSH CRD
        PUSH HANDL
        CALL SetConsoleCursorPosition@8
;задать цветовые атрибуты выводимого текста
        PUSH COL1
        PUSH HANDL
        CALL SetConsoleTextAttribute@8
;вывести строку
        PUSH OFFSET STR1
        CALL LENSTR ;в EBX длина строки
        PUSH 0
        PUSH OFFSET LENS
        PUSH EBX
        PUSH OFFSET STR1
        PUSH HANDL
        CALL WriteConsoleA@20
;ждать ввод строки
        PUSH 0
        PUSH OFFSET LENS
        PUSH 200
        PUSH OFFSET BUF
        PUSH HANDL1
        CALL ReadConsoleA@20
;вывести полученную строку
;вначале задать цветовые атрибуты выводимого текста
        PUSH COL2
```

```
        PUSH HANDL
        CALL SetConsoleTextAttribute@8
;-------------------------------------
        PUSH 0
        PUSH OFFSET LENS
        PUSH [LENS]   ; длина вводимой строки
        PUSH OFFSET BUF
        PUSH HANDL
        CALL WriteConsoleA@20
;небольшая задержка
        MOV ECX,01FFFFFFFH
L1:
        LOOP L1
;закрыть консоль
        CALL FreeConsole@0
        CALL ExitProcess@4
;строка - [EBP+08H]
;длина в EBX
LENSTR PROC
      ENTER 0,0
      PUSH  EAX
      PUSH  EDI
;--------------------
      CLD
      MOV   EDI,DWORD PTR [EBP+08H]
      MOV   EBX,EDI
      MOV   ECX,100   ; ограничить длину строки
      XOR   AL,AL
      REPNE SCASB     ; найти символ 0
      SUB   EDI,EBX   ; длина строки, включая 0
      MOV   EBX,EDI
      DEC   EBX
;--------------------
      POP   EDI
      POP   EAX
      LEAVE
      RET   4
LENSTR ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.3.2:

```
ml /c /coff cons2.asm
link /subsystem:console cons2.obj
```

В программе из листинга 2.3.2, кроме уже описанных функций, появились еще две. `SetConsoleCursorPosition` — установить позицию курсора, и здесь все

довольно ясно: первым параметром идет дескриптор выходного буфера, вторым — структура `COORD`, содержащая координаты курсора. Функция `SetConsoleScreenBufferSize` менее понятна. Она устанавливает размер буфера окна консоли. Этот размер не может уменьшить уже существующий буфер (существующее окно), а может только его увеличить.

Замечу, кстати, что в функции `LENSTR` мы теперь используем пару команд `ENTER` — `LEAVE` *(см. главу 1.2)* вместо обычных сочетаний команд `PUSH EBP` / `MOV EBP,ESP` / `SUB ESP,N` — `MOV ESP,EBP` / `POP EBP`. Честно говоря, никаких особых преимуществ использование команд `ENTER`, `LEAVE` не дает, просто пора расширять свой командный запас.

Обработка событий от мыши и клавиатуры

Данный раздел будет посвящен обработке команд мыши и клавиатуры в консольном приложении. Возможность такой обработки делает консольные приложения весьма гибкими, расширяя круг задач, которые можно решить в этом режиме.

Прежде, однако, мы рассмотрим одну весьма необычную, но чрезвычайно полезную API-функцию. Это функция `wsprintfA`. Я подчеркиваю, что это именно API-функция, которая предоставляется системой приложению. Эта функция является аналогом библиотечной C-функции `sprintf`. Первым параметром функции является указатель на буфер, куда помещается результат форматирования. Второй — указатель на форматную строку, например: `"Числа: %lu, %lu"`. Далее идут указатели на параметры (либо сами параметры, если это числа, см. далее), количество которых определено только содержимым форматной строки. Функция осуществляет копирование форматной строки в буфер, подставляя туда значения параметров. А теперь — самое главное. Поскольку количество параметров не определено, то стек придется освобождать нам. Пример использования этой функции будет дан позже. Если функция выполнена успешно, то в регистр `EAX` будет возвращена длина скопированной строки.

В основе получения информации о клавиатуре и мыши в консольном режиме является функция `ReadConsoleInput`. Параметры этой функции:

- 1-й параметр — дескриптор входного буфера консоли;
- 2-й параметр — указатель на структуру (или массив структур), в которой содержится информация о событиях, происшедших с консолью (далее мы подробно рассмотрим эту структуру);

- 3-й параметр — количество получаемых информационных записей (структур);
- 4-й параметр — указатель на двойное слово, содержащее количество реально полученных записей.

А теперь подробно разберемся со структурой, в которой содержится информация о консольном событии. Прежде всего, замечу, что в С эта структура записывается с помощью типа данных `UNION` *(о типах данных см. главу 2.7)*. На мой взгляд, частое использование этого слова притупляет понимание того, что же за этим стоит. И при описании этой структуры мы обойдемся без слов `STRUCT` и `UNION`. Замечу также, что в начале этого блока данных идет двойное слово, младшее слово которого определяет тип события. В зависимости от значения этого слова последующие байты (максимум 18) будут трактоваться так или иначе. Те, кто уже знаком с различными структурами, используемыми в С и макроассемблере, теперь должны понять, почему `UNION` здесь весьма уместен.

Но вернемся к типу события. Всего системой зарезервировано пять типов событий:

```
KEY_EVENT                  equ 1h   ; клавиатурное событие
MOUSE_EVENT                equ 2h   ; событие с мышью
WINDOW_BUFFER_SIZE_EVENT   equ 4h   ; изменился размер окна
MENU_EVENT                 equ 8h   ; используется системой
FOCUS_EVENT                equ 10h  ; используется системой
```

А теперь разберем значение других байтов структуры в зависимости от происшедшего события.

Событие *KEY_EVENT*

В табл. 2.3.1 приведены описания события `KEY_EVENT` (структура `KEY_EVENT_RECORD`).

Таблица 2.3.1. *Событие `KEY_EVENT`*

Смещение	**Длина**	**Значение**
+4	4	При нажатии клавиши значение поля больше нуля (при условии, если клавиша поддерживается системой)
+8	2	Количество повторов при удержании клавиши
+10	2	Виртуальный код клавиши (машинно-независимый)
+12	2	Скан-код клавиши (генерируемый клавиатурой)

Таблица 2.3.1 *(окончание)*

Смещение	Длина	Значение
+14	2	Для функции `ReadConsoleInputA` — младший байт равен ASCII-коду клавиши. Для функции `ReadConsoleInputW` слово содержит код клавиши в двухбайтовой кодировке Unicode
+16	4	Содержатся состояния управляющих клавиш. Может являться суммой следующих констант: `RIGHT_ALT_PRESSED  equ 1h` `LEFT_ALT_PRESSED   equ 2h` `RIGHT_CTRL_PRESSED equ 4h` `LEFT_CTRL_PRESSED  equ 8h` `SHIFT_PRESSED      equ 10h` `NUMLOCK_ON         equ 20h` `SCROLLLOCK_ON      equ 40h` `CAPSLOCK_ON        equ 80h` `ENHANCED_KEY       equ 100h` Смысл констант очевиден

Событие *MOUSE_EVENT*

В табл. 2.3.2 приведено описание события `MOUSE_EVENT` (структура `MOUSE_EVENT_RECORD`).

Таблица 2.3.2. *Событие `MOUSE_EVENT`*

Смещение	Длина	Значение
+4	4	Младшее слово — *X*-координата курсора мыши, старшее слово — *Y*-координата курсора мыши
+8	4	Описывает состояние кнопок мыши. Первый бит — левая кнопка, второй бит — правая кнопка, третий бит — средняя кнопка. (Напомню, что биты нумеруются с 0.) Если бит установлен в единицу, значит, кнопка нажата
+12	4	Состояние управляющих клавиш (см. табл. 2.3.1 (+16))
+16	4	Может содержать следующие значения: `MOUSE_MOV equ 1h ; было движение мыши` `DOUBLE_CL equ 2h ; был двойной щелчок` `MOUSE_WH  equ 4h ; был поворот колесика мыши`

Событие *WINDOW_BUFFER_SIZE_EVENT*

Рассмотрим событие `WINDOW_BUFFER_SIZE_EVENT`.

По смещению +4 находится двойное слово, содержащее новый размер консольного окна. Младшее слово — это размер по оси *x*, старшее слово — размер по оси *y*. Да, когда речь идет о консольном окне, все размеры и координаты даются в "символьных" единицах.

Что касается последних двух событий, то там также значимым является двойное слово по смещению +4. В листинге 2.3.3 дана простая программа обработки консольных событий.

Листинг 2.3.3. Пример обработки событий от мыши и клавиатуры для консольного приложения

```
;cons3.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE  equ -11
STD_INPUT_HANDLE   equ -10
;тип события
KEY_EV             equ 1h
MOUSE_EV           equ 2h
;константы - состояния клавиатуры
RIGHT_ALT_PRESSED  equ 1h
LEFT_ALT_PRESSED   equ 2h
RIGHT_CTRL_PRESSED equ 4h
LEFT_CTRL_PRESSED  equ 8h
SHIFT_PRESSED      equ 10h
NUMLOCK_ON         equ 20h
SCROLLLOCK_ON      equ 40h
CAPSLOCK_ON        equ 80h
ENHANCED_KEY       equ 100h
;прототипы внешних процедур
EXTERN  wsprintfA:NEAR
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  SetConsoleCursorPosition@8:NEAR
EXTERN  SetConsoleTitleA@4:NEAR
EXTERN  FreeConsole@0:NEAR
EXTERN  AllocConsole@0:NEAR
EXTERN  CharToOemA@8:NEAR
EXTERN  SetConsoleTextAttribute@8:NEAR
```

```
EXTERN  ReadConsoleInputA@16:NEAR
EXTERN  ExitProcess@4:NEAR

;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-------------------------------------------
;структура для определения событий
COOR  STRUC
        X  WORD ?
        Y  WORD ?
COOR  ENDS
;сегмент данных
_DATA SEGMENT
      STR1   DB "Для выхода нажмите ESC",0
      STR2   DB "Обработка событий мыши",0
      HANDL  DWORD ?
      HANDL1 DWORD ?
      BUF    DB 200 dup(?)
      LENS   DWORD ? ;количество выведенных символов
      CO     DWORD ?
      FORM   DB "Координаты: %u %u "
      CRD    COOR <?>
      MOUS_KEY WORD 9 dup(?)
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;перекодировка строк
        PUSH OFFSET STR2
        PUSH OFFSET STR2
        CALL CharToOemA@8
        PUSH OFFSET STR1
        PUSH OFFSET STR1
        CALL CharToOemA@8
;образовать консоль
;вначале освободить уже существующую
        CALL FreeConsole@0
        CALL AllocConsole@0
;получить HANDL1 ввода
        PUSH STD_INPUT_HANDLE
        CALL GetStdHandle@4
        MOV  HANDL1,EAX
;получить HANDL вывода
        PUSH STD_OUTPUT_HANDLE
```

```
        CALL GetStdHandle@4
        MOV  HANDL,EAX
;задать заголовок окна консоли
        PUSH OFFSET STR2
        CALL SetConsoleTitleA@4
;длина строки
        PUSH OFFSET STR1
        CALL LENSTR
;вывести строку
        PUSH 0
        PUSH OFFSET LENS
        PUSH EBX
        PUSH OFFSET STR1
        PUSH HANDL
        CALL WriteConsoleA@20
;цикл ожиданий: движение мыши или двойной щелчок
LOO:
;координаты курсора
        MOV CRD.X,0
        MOV CRD.Y,10
        PUSH CRD
        PUSH HANDL
        CALL SetConsoleCursorPosition@8
;прочитать одну запись о событии
        PUSH OFFSET CO
        PUSH 1
        PUSH OFFSET MOUS_KEY
        PUSH HANDL1
        CALL ReadConsoleInputA@16
;проверим, не с мышью ли что?
        CMP WORD PTR MOUS_KEY,MOUSE_EV
        JNE LOO1
;здесь преобразуем координаты мыши в строку
        MOV AX,WORD PTR MOUS_KEY+6 ;Y-координата курсора мыши
;копирование с обнулением старших битов
        MOVZX EAX,AX
        PUSH  EAX
        MOV   AX,WORD PTR MOUS_KEY+4 ;X-координата курсора мыши
;копирование с обнулением старших битов
        MOVZX EAX,AX
        PUSH EAX
        PUSH OFFSET FORM
        PUSH OFFSET BUF
        CALL wsprintfA
;восстановить стек
        ADD ESP,16
```

```
;перекодировать строку для вывода
        PUSH OFFSET BUF
        PUSH OFFSET BUF
        CALL CharToOemA@8
;длина строки
        PUSH OFFSET BUF
        CALL LENSTR
;вывести на экран координаты курсора
        PUSH 0
        PUSH OFFSET LENS
        PUSH EBX
        PUSH OFFSET BUF
        PUSH HANDL
        CALL WriteConsoleA@20
        JMP  LOO ;к началу цикла
LOO1:
;нет ли события от клавиатуры?
        CMP WORD PTR MOUS_KEY,KEY_EV
        JNE LOO
;есть, какое?
        CMP BYTE PTR MOUS_KEY+14,27
        JNE LOO
;******************************
;закрыть консоль
        CALL FreeConsole@0
        PUSH 0
        CALL ExitProcess@4
        RET
;процедура определения длины строки
;строка - [EBP+08H]
;длина в EBX
LENSTR PROC
      ENTER 0,0
      PUSH  EAX
      PUSH  EDI
      CLD
      MOV   EDI,DWORD PTR [EBP+08H]
      MOV   EBX,EDI
      MOV   ECX,100   ; ограничить длину строки
      XOR   AL,AL
      REPNE SCASB     ; найти символ 0
      SUB   EDI,EBX   ; длина строки, включая 0
      MOV   EBX,EDI
      DEC   EBX
      POP   EDI
```

```
      POP   EAX
      LEAVE
      RET   4
LENSTR ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.3.3:

```
ml /c /coff cons3.asm
link /subsystem:console cons3.obj
```

После того как вы познакомились с программой из листинга 2.3.3, давайте ее подробнее обсудим.

Начнем с функции `wsprintfA`. Как я уже заметил, эта функция необычная. Во-первых, она имеет переменное число параметров. Первые два параметра обязательны. Вначале идет указатель на буфер, куда будет скопирована результирующая строка. Вторым идет указатель на форматную строку. Форматная строка может содержать текст, а также собственно формат выводимых параметров. Поля, содержащие информацию о параметре, начинаются с символа `%`. Формат этих полей в точности соответствует формату полей, используемых в стандартных C-функциях `printf`, `sprintf` и др. Исключением является отсутствие в формате для функции `wsprintfA` вещественных чисел. Нет нужды рассматривать этот формат подробно, замечу только, что каждое поле в форматной строке соответствует параметру (начиная с третьего). В нашем случае форматная строка была равна: `"Координаты: %u %u"`. Это означало, что далее в стек будут отправлены два числовых параметра типа `WORD`. Конечно, в стек мы отправили два двойных слова, позаботившись лишь о том, чтобы старшие слова были обнулены. Для такой операции очень удобна команда микропроцессора `MOVZX`, которая копирует второй операнд в первый так, чтобы биты старшего слова были заполнены нулями. Если бы параметры были двойными словами, то вместо поля `%u` мы бы поставили `%lu`. В случае, если поле форматной строки определяет строку-параметр, например `"%S"`, в стек следует отправлять указатель на строку (что естественно).[2]

Во-вторых, поскольку функция "не знает", сколько параметров может быть в нее отправлено, разработчики не стали усложнять текст этой функции и оставили нам задачу освобождения стека.[3] Стек освобождается командой `ADD ESP,N`. Здесь `N` — это количество освобождаемых байтов.

[2] В этой связи не могу не отметить, что встречал в литературе по ассемблеру (!) утверждение, что все помещаемые в стек для этой функции параметры являются указателями. Как видим, вообще говоря, это не верно.

[3] Компилятор C, естественно, делает это за нас.

Обратимся теперь к функции `ReadConsoleInputA`. К уже сказанному о ней добавлю только то, что если буфер событий пуст, то функция будет ждать, пока "что-то" не случится с консольным окном, и только тогда возвратит управление. Кроме того, мы можем указать, чтобы функция возвращала не одну, а несколько записей о происшедших с консолью событиях. В этом случае в буфер будет помещена не одна, а несколько информационных записей. Но мы на этом останавливаться не будем.

В *приложении 6* и на прилагаемом к книге компакт-диске содержится консольная программа с обработкой всевозможных событий от клавиатуры и мыши. Советую разобрать структуру и функционирование этой поучительной программы.

Таймер в консольном приложении

В последнем разделе главы мы рассмотрим довольно редко освещаемый в литературе вопрос — таймеры в консольном приложении. Надо сказать, что мы несколько опережаем события и рассматриваем таймер в консольном приложении раньше, чем в приложении GUI (Graphic Universal Interface — так называются обычные оконные приложения).

Основным способом создания таймера является использование функции `SetTimer`. Позднее мы будем подробно о ней говорить. Таймер может быть установлен в двух режимах. Первый режим — это когда последний параметр равен нулю. В этом случае на текущее окно (его функцию) через равные промежутки времени, определяемые третьим параметром, будет приходить сообщение `WM_TIMER`. Во втором режиме последний параметр указывает на функцию, которая будет вызываться опять через равные промежутки времени. Однако для консольного приложения эта функция не подходит, т. к. сообщение `WM_TIMER` пересылается окну функцией `DispatchMessage`, которая вызывается в цикле обработки сообщений. Но использование этой функции для консольных приложений проблематично.

Для консольных приложений следует использовать функцию `timeSetEvent`. Вот параметры этой функции:

- 1-й параметр — время задержки таймера, для нас это время совпадает со временем между двумя вызовами таймера;
- 2-й параметр — точность работы таймера (приоритет посылки сообщения);
- 3-й параметр — адрес вызываемой процедуры;
- 4-й параметр — параметр, посылаемый в процедуру;
- 5-й параметр — тип вызова (одиночный или периодический).

Если функция завершилась удачно, то в регистр `EAX` возвращается идентификатор таймера.

Сама вызываемая процедура получает также 5 параметров:

- 1-й параметр — идентификатор таймера;
- 2-й параметр — не используется;
- 3-й параметр — параметр `Dat` (см. `timeSetEvent`);
- 4-й и 5-й параметры — не используются.

Для удаления таймера предназначена функция `timeKillEvent`, параметром которой является идентификатор таймера.

В листинге 2.3.4 вы можете видеть простую программу, использующую таймер.

Листинг 2.3.4. Таймер в консольном режиме

```
;cons4.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE  equ -11
STD_INPUT_HANDLE   equ -10
TIME_PERIODIC      equ 1       ; тип вызова таймера
;атрибуты цветов
FOREGROUND_BLUE    equ 1h      ; синий цвет букв
FOREGROUND_GREEN   equ 2h      ; зеленый цвет букв
FOREGROUND_RED     equ 4h      ; красный цвет букв
FOREGROUND_INTENSITY  equ 8h   ; повышенная интенсивность
BACKGROUND_BLUE    equ 10h     ; синий свет фона
BACKGROUND_GREEN   equ 20h     ; зеленый цвет фона
BACKGROUND_RED     equ 40h     ; красный цвет фона
BACKGROUND_INTENSITY  equ 80h  ; повышенная интенсивность
COL1 = 2h+8h   ; цвет выводимого текста
;прототипы внешних процедур
EXTERN  wsprintfA:NEAR
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  SetConsoleCursorPosition@8:NEAR
EXTERN  SetConsoleTitleA@4:NEAR
EXTERN  FreeConsole@0:NEAR
EXTERN  AllocConsole@0:NEAR
EXTERN  CharToOemA@8:NEAR
EXTERN  SetConsoleCursorPosition@8:NEAR
```

```
EXTERN  SetConsoleTextAttribute@8:NEAR
EXTERN  ReadConsoleA@20:NEAR
EXTERN  timeSetEvent@20:NEAR
EXTERN  timeKillEvent@4:NEAR
EXTERN  ExitProcess@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\winmm.lib
;------------------------------------------------
COOR  STRUC
      X  WORD ?
      Y  WORD ?
COOR  ENDS
;сегмент данных
_DATA SEGMENT
      HANDL  DWORD ?
      HANDL1 DWORD ?
      STR2   DB "Пример таймера в консольном приложении",0
      STR3   DB 100 dup(0)
      FORM   DB "Число вызовов таймера: %lu",0
      BUF    DB 200 dup(?)
      NUM    DWORD 0
      LENS   DWORD ?  ; количество выведенных символов
      CRD    COOR <?>
      ID     DWORD ?  ; идентификатор таймера
      HWND DWORD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;перекодировать строку STR2
        PUSH OFFSET STR2
        PUSH OFFSET STR2
        CALL CharToOemA@8
;образовать консоль
;вначале освободить уже существующую
        CALL FreeConsole@0
        CALL AllocConsole@0
;получить HANDL1 ввода
        PUSH STD_INPUT_HANDLE
        CALL GetStdHandle@4
        MOV  HANDL1,EAX
;получить HANDL вывода
        PUSH STD_OUTPUT_HANDLE
```

```
        CALL GetStdHandle@4
        MOV HANDL,EAX
;задать заголовок окна консоли
        PUSH OFFSET STR2
        CALL SetConsoleTitleA@4
;задать цветовые атрибуты выводимого текста
        PUSH COL1
        PUSH HANDL
        CALL SetConsoleTextAttribute@8
;установить таймер
        PUSH TIME_PERIODIC ; периодический вызов
        PUSH 0
        PUSH OFFSET TIME   ; вызываемая таймером процедура
        PUSH 0             ; точность вызова таймера
        PUSH 1000          ; вызов через одну секунду
        CALL timeSetEvent@20
        MOV ID,EAX
;ждать ввод строки
        PUSH 0
        PUSH OFFSET LENS
        PUSH 200
        PUSH OFFSET BUF
        PUSH HANDL1
        CALL ReadConsoleA@20
;закрыть таймер
        PUSH ID
        CALL timeKillEvent@4
;закрыть консоль
        CALL FreeConsole@0
        PUSH 0
        CALL ExitProcess@4
;строка - [EBP+08H]
;длина в EBX
LENSTR PROC
      ENTER 0,0
      PUSH EAX
      PUSH EDI
;--------------------
      CLD
      MOV   EDI,DWORD PTR [EBP+08H]
      MOV   EBX,EDI
      MOV   ECX,100  ; ограничить длину строки
      XOR   AL,AL
      REPNE SCASB    ; найти символ 0
      SUB EDI,EBX    ; длина строки, включая 0
```

```
      MOV EBX,EDI
      DEC EBX
;--------------------
      POP EDI
      POP EAX
      LEAVE
      RET 4
LENSTR ENDP
;процедура вызывается таймером
TIME  PROC
      PUSHAD   ; сохранить все регистры
;установить позицию курсора
      MOV  CRD.X,0
      MOV  CRD.Y,10
      PUSH CRD
      PUSH HANDL
      CALL SetConsoleCursorPosition@8
;заполнить строку STR3
      PUSH NUM
      PUSH OFFSET FORM
      PUSH OFFSET STR3
      CALL wsprintfA
      ADD  ESP,12   ; восстановить стек
;перекодировать строку STR3
      PUSH OFFSET STR3
      PUSH OFFSET STR3
      CALL CharToOemA@8
;вывести строку с номером вызова таймера
      PUSH OFFSET STR3
      CALL LENSTR
      PUSH 0
      PUSH OFFSET LENS
      PUSH EBX
      PUSH OFFSET STR3
      PUSH HANDL
      CALL WriteConsoleA@20
      INC NUM
      POPAD
      RET 20   ; выход с освобождением стека
TIME  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.3.4:

```
ml /c /coff cons4.asm
link /subsystem:console cons4.obj
```

Программа в листинге 2.3.4 будет выводить в окно значение счетчика, которое каждую секунду увеличивается на единицу.

Я начал данную главу с рассуждения о командной строке, но до сих пор не объявил, как работать с командной строкой. О, здесь все очень просто. Есть API-функция `GetCommandLine`, которая возвращает указатель на командную строку. Эта функция одинаково работает как для консольных приложений, так и для приложений GUI. Далее представлена программа (листинг 2.3.5), печатающая параметры командной строки. Надеюсь, вы понимаете, что первым параметром является полное имя программы.

Листинг 2.3.5. Пример работы с параметрами командной строки

```
;cons5.asm
;программа вывода параметров командной строки
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
;прототипы внешних процедур
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetCommandLineA@0:NEAR
EXTERN  CharToOemA@8:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      BUF   DB  100 dup(0)
      LENS  DWORD ?   ; количество выведенных символов
      NUM   DWORD ?
      CNT   DWORD ?
      HANDL DWORD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить HANDLE вывода
        PUSH STD_OUTPUT_HANDLE
        CALL GetStdHandle@4
        MOV  HANDL,EAX
```

```
;получить количество параметров
        CALL NUMPAR
        MOV  NUM,EAX
        MOV  CNT,0
;-------------------------------
;вывести параметры командной строки
LL1:
        MOV EDI,CNT
        CMP NUM,EDI
        JE  LL2
;номер параметра
        INC EDI
        MOV CNT,EDI
;получить параметр с номером EDI
        LEA  EBX,BUF
        CALL GETPAR
;получить длину параметра
        PUSH OFFSET BUF
        CALL LENSTR
;в конце - перевод строки
        MOV BYTE PTR [BUF+EBX],13
        MOV BYTE PTR [BUF+EBX+1],10
        MOV BYTE PTR [BUF+EBX+2],0
        ADD EBX,2
;вывод строки
;может потребоваться перекодировка
        PUSH OFFSET BUF
        PUSH OFFSET BUF
        CALL CharToOemA@8
        PUSH 0
        PUSH OFFSET LENS
        PUSH EBX
        PUSH OFFSET BUF
        PUSH HANDL
        CALL WriteConsoleA@20
        JMP LL1
LL2:
        PUSH 0
        CALL ExitProcess@4
;строка - [EBP+08H]
;длина в EBX
LENSTR PROC
      PUSH EBP
      MOV EBP,ESP
      PUSH EAX
      PUSH EDI
```

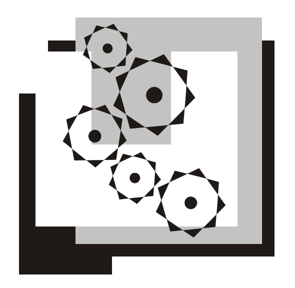
```
;--------------------
      CLD
      MOV   EDI,DWORD PTR [EBP+08H]
      MOV   EBX,EDI
      MOV   ECX,100   ; ограничить длину строки
      XOR   AL,AL
      REPNE SCASB     ; найти символ 0
      SUB   EDI,EBX   ; длина строки, включая 0
      MOV   EBX,EDI
      DEC   EBX
;--------------------
      POP EDI
      POP EAX
      POP EBP
      RET 4
LENSTR ENDP
;определить количество параметров (->EAX)
NUMPAR PROC
      CALL GetCommandLineA@0
      MOV ESI,EAX   ; указатель на строку
      XOR ECX,ECX   ; счетчик
      MOV EDX,1     ; признак
L1:
      CMP BYTE PTR [ESI],0
      JE  L4
      CMP BYTE PTR [ESI],32
      JE  L3
      ADD ECX,EDX   ; номер параметра
      MOV EDX,0
      JMP L2
L3:
      OR EDX,1
L2:
      INC ESI
      JMP L1
L4:
      MOV EAX,ECX
      RET
NUMPAR ENDP
;получить параметр
;EBX - указывает на буфер, куда будет помещен параметр
;в буфер помещается строка с нулем на конце
;EDI - номер параметра
GETPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX  ; указатель на строку
```

```
      XOR  ECX,ECX  ; счетчик
      MOV  EDX,1    ; признак
L1:
      CMP BYTE PTR [ESI],0
      JE  L4
      CMP BYTE PTR [ESI],32
      JE  L3
      ADD ECX,EDX   ; номер параметра
      MOV EDX,0
      JMP L2
L3:
      OR EDX,1
L2:
      CMP ECX,EDI
      JNE L5
      MOV AL,BYTE PTR [ESI]
      MOV BYTE PTR [EBX],AL
      INC EBX
L5:
      INC ESI
      JMP L1
L4:
      MOV BYTE PTR [EBX],0
      RET
GETPAR ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.3.5:

```
ml /c /coff cons5.asm
link /subsystem:console cons5.obj
```

Рекомендую читателю разобраться в алгоритме работы процедур `NUMPAR` и `GETPAR` (из листинга 2.3.5).



Глава 2.4

Понятие ресурса. Редакторы и трансляторы ресурсов

Операционной системой Windows поддерживаются такие объекты, как *ресурсы*. Английское слово resource можно перевести и как запас. Так вот, это некоторый запас, хранящийся в исполняемом модуле, который может быть использован для построения диалоговых окон. Ресурс представляет собой некий визуальный элемент (хотя и не всегда) с заданными свойствами, хранящийся в исполняемом файле отдельно от кода и данных, и может отображаться специальными API-функциями. Использование ресурсов дает два вполне определенных преимущества:

- ресурсы загружаются в память лишь при обращении к ним, тем самым экономится память;
- свойства ресурсов, их загрузка и отображение поддерживаются системой автоматически, не требуя от программиста написания дополнительного кода.

Описание ресурсов хранится отдельно от программы в текстовом файле с расширением rc и компилируется в объектный модуль с расширением res специальным транслятором ресурсов. Транслятором ресурсов в пакете MASM32 является утилита RC.EXE. Кроме этого, в пакете MASM32 версии 9.0 имеется еще один транслятор ресурсов PORC.EXE. После того как ресурс откомпилирован, он может быть включен в исполняемый модуль обычным редактором связей LINK.EXE.

Язык описания ресурсов

В данном разделе мы займемся изучением языка ресурсов, который и по синтаксису, и по своей семантике в значительной степени напоминает классический язык С. Зная его, можно вполне обойтись без специального редактора

ресурсов. В настоящее время существует большое количество редакторов ресурсов. Видимо, поэтому книги по программированию уделяют мало внимания языку описания ресурсов. Мы, напротив, не будем касаться этих программ[1], а подробнее остановимся на структуре ресурсных файлов и языке ресурсов.

Начнем с перечисления наиболее употребляемых ресурсов:

- пиктограммы;
- курсоры;
- битовая картинка;
- строка;
- диалоговое окно;
- меню;
- акселераторы.

Вот наиболее распространенные ресурсы. Надо только иметь в виду, что такой ресурс, как диалоговое окно, может содержать в себе управляющие элементы, которые также должны быть определены, но в рамках описания окна. Но об этом поговорим несколько позже.

Пиктограммы

Пиктограммы (icons) хранятся в отдельном файле с расширением ico. Ссылка на этот файл должна указываться в файле ресурсов. Вот фрагмент файла ресурсов resu.rc с описанием пиктограммы.

```
#define IDI_ICON1      1
IDI_ICON1 ICON "Cdrom01.ico"
```

Как видите, фрагмент содержит всего две значимых строки. Одна строка определяет идентификатор пиктограммы, вторая — ассоциирует идентификатор с файлом Cdrom01.ico. Оператор `define` является C-оператором препроцессора. Как я уже упоминал, язык ресурсов очень напоминает язык C. Откомпилируем текстовый файл resu.rc, запустив командную строку: `RC resu.rc`. На диске появляется объектный файл resu.res. При компоновке укажем этот файл в командной строке:

```
LINK /subsystem:windows resu.obj resu.res
```

Таким образом, файл ресурсов будет объединен с обычным объектным файлом и в результате будет получен исполняемый файл, содержащий ресурс "пиктограмма".

[1] Лично я обычно предпочитаю использовать редактор ресурсов из пакета Visual Studio .NET либо простой текстовый редактор.

У читателя возникает вопрос: как использовать данный ресурс во время выполнения программы? Здесь все просто: предположим, что мы хотим установить новую пиктограмму для окна. Вот фрагмент программы, который устанавливает стандартную пиктограмму для главного окна:

```
PUSH  IDI_APPLICATION
PUSH  0
CALL  LoadIconA@8
MOV  WC.CLSHICON, EAX
```

А вот фрагмент программы для установки пиктограммы, указанной в файле ресурсов:

```
PUSH  1       ; идентификатор пиктограммы (см. файл resu.rc)
PUSH  HINST   ; идентификатор процесса
CALL  LoadIconA@8
MOV  WC.CLSHICON, EAX
```

Итак, для того чтобы использовать пиктограмму в своем приложении, необходимо:

1. Создать ее в каком-либо подходящем редакторе.
2. Включить ссылку на файл пиктограммы в файле описания ресурсов.
3. Использовать пиктограмму, предварительно загрузив ее при помощи API-функции `LoadIcon`.

Курсоры

Подход здесь полностью идентичен. Привожу ниже файл ресурсов, где определены и курсор, и пиктограмма:

```
#define IDI_ICON1     1
#define IDI_CUR1      2

IDI_ICON1 ICON "Cdrom01.ico"
IDI_CUR1 CURSOR "4way01.cur"
```

А вот фрагмент программы, вызывающей пиктограмму и курсор:

```
;пиктограмма окна
      PUSH  1 ;идентификатор пиктограммы
      PUSH  HINST
      CALL  LoadIconA@8
      MOV   WC.CLSHICON, EAX
;курсор окна
      PUSH  2 ;идентификатор курсора
      PUSH  HINST
      CALL  LoadCursorA@8
      MOV   WC.CLSHCURSOR, EAX
```

Битовые изображения

Здесь ситуация аналогична двум предыдущим. Вот пример фрагмента файла ресурсов с битовой картинкой:

```
#define BIT1 1
BIT1 BITMAP "PIR2.BMP"
```

Строки

Чтобы задать строку или несколько строк, используется ключевое слово `STRINGTABLE`. Далее представлен текст ресурса, задающий две строки. Для загрузки строки в программу используется функция `LoadString` (см. ниже). Строки, задаваемые в файле ресурсов, могут играть роль констант.

```
#define STR1 1
#define STR2 2

STRINGTABLE
{
  STR1, "Сообщение"
  STR2, "Версия 1.01"
}
```

Диалоговые окна

Диалоговые окна (или просто диалоги) являются наиболее сложными элементами ресурсов. В отличие от ресурсов, которые мы до сих пор рассматривали, для диалога не задается идентификатор. Обращение к диалогу происходит по его имени (строке). Рассмотрим следующий фрагмент:

```
#define WS_SYSMENU       0x00080000L
#define WS_MINIMIZEBOX   0x00020000L
#define WS_MAXIMIZEBOX   0x00010000L

DIAL1 DIALOG 0, 0, 240, 120
STYLE WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
CAPTION "Пример диалогового окна"
FONT 8, "Arial"
{
}
```

Как видим, определение диалога начинается со строки, содержащей ключевое слово `DIALOG`. В этой же строке далее указывается положение и размер диалогового окна. Затем идут строки, содержащие другие свойства окна. Наконец, идут фигурные скобки. В данном случае они пусты. Это означает, что в окне нет никаких управляющих элементов. Тип окна, а также других

элементов определяется константами, которые мы поместили в начале файла. Эти константы стандартны, и в пакете Visual C++ хранятся в файле RESOURCE.H. Мы же, как и раньше, все константы будем определять непосредственно в файле ресурсов. Обращаю ваше внимание, что константы определяются согласно нотации языка C.

Прежде чем разбирать пример в листинге 2.4.1, рассмотрим особенности работы с диалоговыми окнами. Диалоговое окно очень похоже на обычное окно. Так же как обычное окно, оно имеет свою процедуру обработки сообщений. Процедура диалогового окна имеет те же входные параметры, что и процедура обычного окна. Сообщений, которые приходят на процедуру диалогового окна, гораздо меньше. Но те, которые у диалогового окна имеются, в основном совпадают с аналогичными сообщениями для обычного окна. Только вместо сообщения `WM_CREATE` приходит сообщение `WM_INITDIALOG`. Процедура диалогового окна может возвращать либо нулевое, либо ненулевое значение. Ненулевое значение должно возвращаться в том случае, если процедура обрабатывает (берет на себя обработку) данное сообщение, и ноль, если предоставляет обработку системе.

Отличия в поведении диалогового окна от обычного окна легко объяснить. Действительно, если вы создаете обычное окно, то все его свойства определяются тремя факторами: свойствами класса, свойствами, определяемыми при создании окна, реакцией процедуры окна на определенные сообщения. При создании диалогового окна все свойства заданы в описании ресурса. Часть этих свойств задается, когда при вызове функции создания диалогового окна (`DialogBox`, `DialogBoxParam` и др.) неявно вызывается функция `CreateWindow`. Остальная же часть свойств определяется поведением внутренней функции, которую порождает система при создании диалогового окна. Если с диалоговым окном что-то происходит, то сообщение сначала приходит на внутреннюю процедуру, а затем вызывается процедура диалогового окна, которую мы создаем в программе. Если процедура возвращает 0, то внутренняя процедура продолжает обработку данного сообщения, если же возвращается ненулевое значение, внутренняя процедура не обрабатывает сообщение. Вот, вкратце, как работают механизмы, регулирующие работу диалогового окна. Рассмотрите теперь программу в листинге 2.4.1, после которого дано ее разъяснение.

Листинг 2.4.1. Демонстрация использования простых ресурсов

```
//файл dial.rc
//определение констант
#define WS_SYSMENU     0x00080000L
#define WS_MINIMIZEBOX   0x00020000L
```

```
#define WS_MAXIMIZEBOX   0x00010000L

//идентификаторы
#define STR1 1
#define STR2 2
#define IDI_ICON1 3

//определили пиктограмму
IDI_ICON1 ICON "icon.ico"

//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 120
STYLE WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
CAPTION "Пример диалогового окна"
FONT 8, "Arial"
{
}

//определение строк
STRINGTABLE
{
 STR1, "Сообщение"
 STR2, "Версия программы 1.00"
}
;файл dialog1.inc
;константы
;сообщение приходит при закрытии окна
WM_CLOSE       equ 10h
WM_INITDIALOG  equ 110h
WM_SETICON     equ 80h
;прототипы внешних процедур
EXTERN  MessageBoxA@16:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  LoadStringA@16:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  SendMessageA@16:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND    DD ?
      MSMESSAGE DD ?
      MSWPARAM  DD ?
      MSLPARAM  DD ?
```

```
      MSTIME    DD ?
      MSPT      DD ?
MSGSTRUCT ENDS

;файл dial.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include dialog1.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
; -----------------------------------------------------
; сегмент данных
_DATA SEGMENT
      MSG       MSGSTRUCT <?>
      HINST     DD 0 ;дескриптор приложения
      PA        DB "DIAL1",0
      BUF1      DB 40 dup(0)
      BUF2      DB 40 dup(0)
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV  [HINST], EAX
;-------------------------------
;--загрузить строку
      PUSH 40
      PUSH OFFSET BUF1
      PUSH 1
      PUSH [HINST]
      CALL LoadStringA@16
;--загрузить строку
      PUSH 40
      PUSH OFFSET BUF2
      PUSH 2
      PUSH [HINST]
      CALL LoadStringA@16
;-------------------------------
      PUSH 0          ;MB_OK
      PUSH OFFSET BUF1
      PUSH OFFSET BUF2
      PUSH 0
```

```
      CALL MessageBoxA@16
;создать диалоговое окно
      PUSH 0
      PUSH OFFSET WNDPROC ; процедура окна
      PUSH 0
      PUSH OFFSET PA      ; название ресурса (DIAL1)
      PUSH [HINST]
      CALL DialogBoxParamA@20
      CMP  EAX,-1
      JNE  KOL
KOL:
;----------------------------------
      PUSH 0
      CALL ExitProcess@4
;----------------------------------
;процедура диалогового окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH  EBP
      MOV   EBP,ESP
      PUSH  EBX
      PUSH  ESI
      PUSH  EDI
;----------------
      CMP   DWORD PTR [EBP+0CH],WM_CLOSE
      JNE   L1
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP FINISH
L1:
      CMP   DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE   FINISH
;загрузить пиктограмму
      PUSH  3       ; идентификатор пиктограммы
      PUSH [HINST]  ; идентификатор процесса
      CALL LoadIconA@8
;установить пиктограмму
        PUSH  EAX
        PUSH  0  ; тип пиктограммы (маленькая)
        PUSH  WM_SETICON
        PUSH  DWORD PTR [EBP+08H]
```

```
        CALL  SendMessageA@16
FINISH:
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      MOV EAX,0
      RET 16
WNDPROC ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.4.1:

```
ml /c /coff dial.asm
rc  dial.rc
link /subsystem:windows dial.obj dial.res
```

Рассмотрим теперь, как работает программа из листинга 2.4.1.

Файл ресурсов должен быть вам понятен, т. к. все используемые там ресурсы были подробно рассмотрены ранее. Замечу только, что файл ресурсов содержит сразу несколько элементов. При этом все ресурсы, кроме диалогового окна, должны иметь идентификатор. Для диалогового окна определяющим является его название, в нашем случае это `DIAL1`.

Перед тем как вызвать диалоговое окно, демонстрируется, как нужно работать с таким ресурсом, как строка. Как видите, это достаточно просто. При помощи функции `LoadString` строка загружается в буфер, после чего с ней можно работать, как с обычной строкой.

Вызов диалогового окна достаточно очевиден, так что перейдем сразу к процедуре диалогового окна. Начнем с сообщения `WM_INITDIALOG`. Это сообщение, как и сообщение `WM_CREATE` для обычного окна, приходит один раз при создании окна. Это весьма удобно для выполнения некоторых операций в самом начале, иными словами, для начальной инициализации. Мы используем эту возможность для определения пиктограммы диалогового окна. Вначале загружаем пиктограмму, а далее посылаем сообщение "установить пиктограмму" для данного окна (`WM_SETICON`). Вторым сообщением, которое мы обрабатываем, является `WM_CLOSE`. Это сообщение приходит, когда происходит щелчок мышью по крестику в правом верхнем углу экрана. После получения этого сообщения выполняется функция `EndDialog`, что приводит к удалению диалогового окна из памяти, выходу из функции `DialogBoxParamA` и в конечном итоге — к выходу из программы.

Ранее было сказано, что процедура диалогового окна должна возвращать ненулевое значение, если она берет на себя обработку данного сообщения.

Как видно из данного примера, в принципе в этом не всегда есть необходимость. В дальнейшем мы акцентируем внимание на тех случаях, когда это потребуется.

> ***Замечание***
>
> Диалоговое окно обладает некоторыми интересными свойствами. Например, при нажатии клавиши <Esc> диалоговому окну (процедуре окна) посылается сообщение `WM_COMMAND` с идентификационным номером 2 (`IDCANCEL`) — имитация нажатия кнопки **Cancel**. Имейте это в виду, когда присваиваете идентификаторам элементов управления окна какие-либо числовые значения.

Меню

Меню также может быть задано в файле ресурсов. Как и диалоговое окно, в программе оно определяется по имени (строке). Меню можно задать и в обычном окне, и в диалоговом окне. Для обычного окна при регистрации класса следует просто заменить строку (когда мы регистрируем класс окон)

```
MOV   DWORD PTR WC.CLMENNAME,0
```

на

```
MOV   DWORD PTR WC.CLMENNAME,OFFSET MENS
```

Здесь `MENS` — имя, под которым меню располагается в файле ресурсов.

Меню на диалоговое окно устанавливается другим способом[2], который, разумеется, подходит и для обычного окна. Вначале меню загружается при помощи функции `LoadMenu`, а затем устанавливается функцией `SetMenu`.

А теперь обо всем подробнее. Рассмотрим структуру файла ресурсов, содержащего определение меню. Ниже представлен текст файла, содержащего определение меню, а затем представлена программа, демонстрирующая меню на диалоговом окне.

```
MENUP MENU
{
 POPUP "&Первый пункт"
 {
   MENUITEM "&Первый", 1
   MENUITEM "В&торой",2
   POPUP "Подмен&ю"
   {
     MENUITEM "Десятый пунк&т", 6
   }
```

[2] Разница между первым и вторым способом задания меню в окне заключается в том, что в первом случае меню задается для целого класса окон, а во втором — для одного конкретного окна.

```
 }
 POPUP "&Второй пункт"
 {
    MENUITEM "Трети&й",3
    MENUITEM "Четверт&ый",4
 }
 MENUITEM "Вы&ход", 5
}
```

Внимательно рассмотрите текст меню. Как видите, пункты меню имеют идентификаторы, по которым в программе можно определить, какой пункт меню выбран. Можно заметить, что выпадающее меню может содержать еще и подменю. Обратите внимание на наличие в строках меню знака `&` — амперсанд. Он означает, что следующий за ним символ при отображении меню будет подчеркнут. Видимый же пункт меню подчеркнутым символом может быть вызван командой <Alt>+<символ с подчеркиванием>.

Рассмотрим пример, представленный в листинге 2.4.2.

Листинг 2.4.2. Пример программы с меню

```
//файл menu.rc
//определение констант
#define WS_SYSMENU       0x00080000L
#define WS_MINIMIZEBOX   0x00020000L
#define WS_MAXIMIZEBOX   0x00010000L
#define WS_POPUP         0x80000000L
#define WS_CAPTION       0x00C00000L

MENUP MENU
{
 POPUP "&Первый пункт"
 {
   MENUITEM "&Первый", 1
   MENUITEM "В&торой",2

 }
 POPUP "&Второй пункт"
 {
   MENUITEM "Трети&й",3
   MENUITEM "Четверт&ый",4
   POPUP "Еще подмен&ю"
   {
    MENUITEM "Десятый пунк&т", 6
   }
 }
```

```
 MENUITEM "Вы&ход", 5
}

//идентификаторы
#define IDI_ICON1 100

//определили пиктограмму
IDI_ICON1 ICON "icon.ico"

//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 120
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
CAPTION "Пример диалогового окна"
FONT 8, "Arial"
{
}
;файл menu.inc
;константы
;сообщение приходит при закрытии окна
WM_CLOSE       equ 10h
WM_INITDIALOG  equ 110h
WM_SETICON     equ 80h
WM_COMMAND     equ 111h
;прототипы внешних процедур
EXTERN  MessageBoxA@16:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  LoadStringA@16:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  LoadMenuA@8:NEAR
EXTERN  SendMessageA@16:NEAR
EXTERN  SetMenu@8:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;файл menu.asm
.586P
```

```
;плоская модель памяти
.MODEL FLAT, stdcall
include menu.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG     MSGSTRUCT <?>
      HINST   DD 0 ;дескриптор приложения
      PA      DB "DIAL1",0
      PMENU   DB "MENUP",0
      STR1    DB "Выход из программы",0
      STR2    DB "Сообщение",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV  [HINST], EAX
;--------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  HINST
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
KOL:
      PUSH  0
      CALL  ExitProcess@4
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  LPARAM
; [EBP+10H]   WAPARAM
; [EBP+0CH]   MES
; [EBP+8]     HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
```

```
      PUSH EDI
;----------------
      CMP DWORD PTR [EBP+0CH],WM_CLOSE
      JNE L1
;закрыть диалоговое окно
        PUSH 0
        PUSH DWORD PTR [EBP+08H]
        CALL EndDialog@8
        JMP  FINISH
L1:
        CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
        JNE  L2
;загрузить пиктограмму
        PUSH  100    ; идентификатор пиктограммы
        PUSH  HINST  ; идентификатор процесса
        CALL  LoadIconA@8
;установить пиктограмму
        PUSH EAX
        PUSH 0  ;тип пиктограммы (маленькая)
        PUSH WM_SETICON
        PUSH DWORD PTR [EBP+08H]
        CALL SendMessageA@16
;загрузить меню
        PUSH OFFSET PMENU
        PUSH HINST
        CALL LoadMenuA@8
;установить меню
        PUSH EAX
        PUSH DWORD PTR [EBP+08H]
        CALL SetMenu@8
        JMP FINISH
L2:
;проверяем, не случилось ли чего с управляющими
;элементами на диалоговом окне
;в нашем случае имеется единственный управляющий
;элемент - это меню
        CMP DWORD PTR [EBP+0CH],WM_COMMAND
        JNE FINISH
;здесь определяем идентификатор, в данном случае
;это идентификатор пункта меню
        CMP WORD PTR [EBP+10H],5
        JNE FINISH
;сообщение
        PUSH 0         ;MB_OK
        PUSH OFFSET STR2
        PUSH OFFSET STR1
```

```
        PUSH 0
        CALL MessageBoxA@16
;закрыть диалоговое окно
        PUSH 0
        PUSH DWORD PTR [EBP+08H]
        CALL EndDialog@8
FINISH:
        MOV EAX,0
        POP EDI
        POP ESI
        POP EBX
        POP EBP
        RET 16
WNDPROC  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.4.2:

```
ml /c /coff menu.asm
rc  menu.rc
link /subsystem:windows menu.obj menu.res
```

Дадим небольшой комментарий к программе из листинга 2.4.2. Прежде всего, обращаю ваше внимание на довольно прозрачную аналогию между диалоговым окном и меню. Действительно, и в том и в другом случае ресурс определяется не по идентификатору, а по имени. Далее, и диалоговое окно, и меню содержат в себе элементы, определяющиеся по их идентификаторам, которые мы задали в файле ресурсов и которые помещаются в младшее слово параметра `WPARAM`.

В программе из листинга 2.4.2 мы программно загружаем меню. Можно поступить и еще одним способом: указать меню в опциях определения диалогового окна следующим образом.

```
//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 120
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
MENU MENUP
CAPTION "Пример диалогового окна"
FONT 8, "Arial"
{
}
```

Этого достаточно, чтобы меню загрузилось и отобразилось автоматически.

Хочу напомнить читателю одну вещь. В *главе 1.3* (см. листинг 1.3.1) приводился пример кнопки, которая создавалась как дочернее окно. То, что нажата

именно эта кнопка, мы определяли по содержимому параметра LPARAM, который содержал дескриптор кнопки. Как видите, идентифицировать элемент, расположенный на диалоговом окне, можно и по дескриптору, и по идентификатору ресурса.

Вернемся к меню. Пункты меню могут содержать дополнительные параметры, которые определяют дополнительные свойства этих пунктов. Вот эти свойства, понимаемые компилятором ресурсов:

- CHECKED — пункт отмечен галочкой;
- GRAYED — элемент недоступен (имеет серый цвет);
- HELP — элемент может быть связан со справочной системой. Редакторы ресурсов дополнительно создают ресурс — строку. При этом идентификатор строки совпадает с идентификатором пункта меню;
- MENUBARBREAK — для горизонтального пункта это означает, что, начиная с него, горизонтальные пункты располагаются в новой строке. Для вертикального пункта — то, что, начиная с него, пункты расположены в новом столбце. При этом проводится разделительная линия;
- MENUBREAK — аналогично предыдущему, но разделительная линия не проводится;
- INACTIVE — пункт не срабатывает;
- SEPARATOR — в меню добавляется разделитель. При этом идентификатор не ставится.

Заканчивая рассуждение о меню, замечу, что у Windows есть обширнейший набор функций, с помощью которых можно менять свойства меню (удалять и добавлять пункты меню, менять их свойства). В следующей главе приводятся примеры некоторых функций этой группы.

Акселераторы

На первый взгляд этот вопрос достаточно прост, но, как станет ясно, он потянет за собой множество других. Акселератор позволяет выбирать пункт меню просто сочетанием клавиш. Это очень удобно и быстро. Таблица акселераторов является ресурсом, имя которого должно совпадать с именем того меню (ресурса), пункты которого она определяет. Вот пример такой таблицы. Определяется один акселератор на пункт меню MENUP, имеющий идентификатор 4:

```
MENUP ACCELERATORS
{
  VK_F5, 4, VIRTKEY
}
```

А вот общий вид таблицы акселераторов:

```
Имя ACCELERATORS
{
 Клавиша 1, Идентификатор пункта меню (1) [,тип][,параметр]
 Клавиша 2, Идентификатор пункта меню (2) [,тип][,параметр]
 Клавиша 3, Идентификатор пункта меню (3) [,тип][,параметр]
 ...
 Клавиша N, Идентификатор пункта меню (N) [,тип][,параметр]
}
```

Рассмотрим представленную схему. Клавиша — это либо символ в кавычках, либо ASCII-код символа, либо виртуальная клавиша. Если вначале стоит код символа, то тип задается как ASCII. Если используется виртуальная клавиша, то тип определяется как `VIRTUAL`. Все названия (макроимена) виртуальных клавиш можно найти в include-файлах (windows.inc). Мы, как обычно, будем определять все макроимена непосредственно в программе.

Параметр может принимать одно из следующих значений: `NOINVERT`, `ALT`, `CONTROL`, `SHIFT`. Значение `NOINVERT` означает, что не подсвечивается выбранный при помощи акселератора пункт меню. Значения `ALT`, `SHIFT`, `CONTROL` означают, что, кроме клавиши, определенной в акселераторе, должна быть нажата одна из управляющих клавиш. Кроме этого, если клавиша определяется в кавычках, то нажатие при этом клавиши `CONTROL` определяется знаком `^`, например, так `^A`.

А теперь поговорим о механизме работы акселераторов. Для того чтобы акселераторы работали, необходимо выполнить два условия:

- должна быть загружена таблица акселераторов. Для этого используется функция `LoadAccelerators`;
- сообщения, пришедшие от акселератора, следует преобразовать в сообщение `WM_COMMAND`. Здесь нам пригодится функция `TranslateAccelerator`.

Остановимся подробнее на втором пункте. Функция `TranslateAccelerator` преобразует сообщения `WM_KEYDOWN` и `WM_SYSKEYDOWN` в сообщения `WM_COMMAND` и `WM_SYSCOMMAND` соответственно. При этом в старшем слове параметра `WPARAM` помещается 1, как отличие для акселератора. В младшем слове, как вы помните, содержится идентификатор пункта меню. Возникает вопрос: для чего необходимы два сообщения: `WM_COMMAND` и `WM_SYSCOMMAND`? Здесь все закономерно: сообщение `WM_SYSCOMMAND` генерируется для пунктов системного меню или меню окна (см. рис. 2.4.1, где представлены меню, которые могут присутствовать в диалоговом окне).

Функция `TranslateAccelerator` возвращает ненулевое значение, если было произведено преобразование сообщения акселератора, в противном случае

возвращается 0. Естественно включить вызов этой функции в цикл обработки сообщений. Вот этот фрагмент.

```
MSG_LOOP:
        PUSH    0
        PUSH    0
        PUSH    0
        PUSH    OFFSET MSG
        CALL    GetMessageA@16
        CMP     EAX, 0
        JE      END_LOOP
        PUSH    OFFSET MSG
        PUSH    ACC
        PUSH    NEWHWND
        CALL    TranslateAcceleratorA@12
        CMP     EAX,0
        JNE     MSG_LOOP
        PUSH    OFFSET MSG
        CALL    TranslateMessage@4
        PUSH    OFFSET MSG
        CALL    DispatchMessageA@4
        JMP     MSG_LOOP
END_LOOP:
```

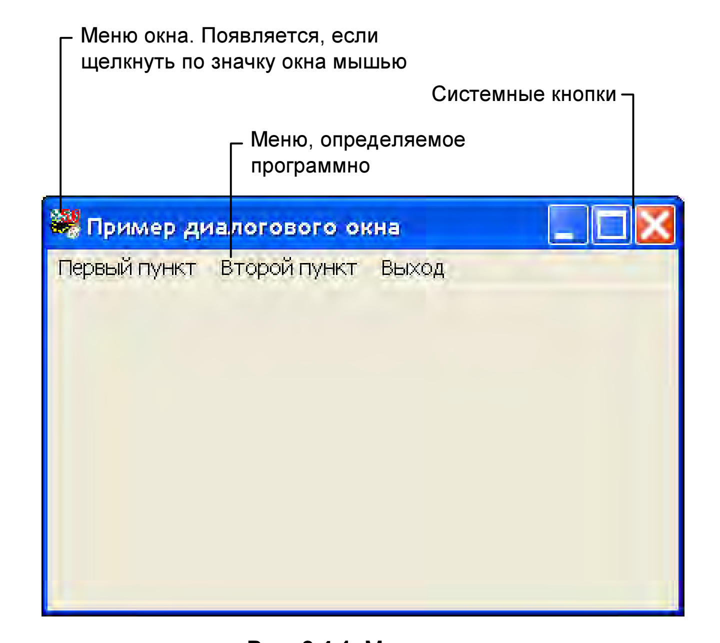


Рис. 2.4.1. Меню окна

Фрагмент вам знаком, но в него вставлена функция `TranslateAccelerator`. Первым параметром этой функции идет дескриптор приложения, вторым параметром идет дескриптор таблицы акселераторов (`[ACC]`), получаемый при загрузке таблицы с помощью функции `LoadAccelerators`. Третий параметр — адрес, где содержится сообщение, полученное функцией `GetMessage`.

И вот тут начинается самое интересное, потому что вы можете сказать, что цикл обработки сообщений используется, когда основное окно создается обычным способом, посредством регистрации класса окна и последующего вызова функции `CreateWindow`. В данной же главе мы рассматриваем диалоговые окна. Конечно, диалоговые окна могут порождаться обычным окном, и здесь все нормально. Но как быть в простейшем случае одного диалогового окна? Первое, что приходит в голову, — это поместить функцию `TranslateAccelerator` в функцию окна и там, на месте, осуществлять преобразования. Но сообщения от акселератора до этой функции не доходят. И здесь мы подходим к новому материалу: модальные и немодальные окна.

Немодальные диалоговые окна

До сих пор мы работали с модальными диалоговыми окнами. Эти окна таковы, что при их вызове программа должна дожидаться, когда окно будет закрыто. Существуют и немодальные диалоговые окна. После их вызова программа продолжает свою работу. При этом немодальное окно позволяет переключаться на другие окна. Таким образом, те окна, которые мы создавали в предыдущих разделах, по сути, являлись немодальными окнами. Каковы особенности создания немодального диалога?

- Немодальный диалог создается при помощи функции `CreateDialog`.
- Уничтожается немодальный диалог функцией `DestroyWindow`.
- Для того чтобы немодальный диалог появился на экране, нужно либо указать у него свойство `WS_VISIBLE`, либо после создания диалога выполнить команду `ShowWindow`.

В листинге 2.4.3 представлена программа, демонстрирующая немодальный диалог (рис. 2.4.2) с меню и обработкой сообщений акселератора. Если немодальное окно является главным, то, как для обычного окна, для него придется создавать цикл обработки сообщений. В качестве акселераторной клавиши предполагается функциональная клавиша <F5> (см. файл ресурсов).

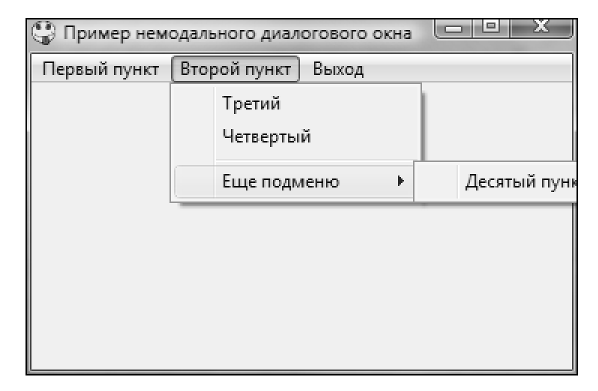


Рис. 2.4.2. Немодальное диалоговое окно

Листинг 2.4.3. Пример немодального диалогового окна с меню и обработкой сообщений акселераторов

```
//файл menu1.rc
//определение констант
#define WS_SYSMENU       0x00080000L
#define WS_MINIMIZEBOX   0x00020000L
#define WS_MAXIMIZEBOX   0x00010000L
#define WS_POPUP         0x80000000L
#define VK_F5            0x74
#define st WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX

MENUP MENU
{
 POPUP "&Первый пункт"
 {
 MENUITEM "&Первый", 1
 MENUITEM "В&торой", 2, HELP
 MENUITEM "Что-то?", 8
 }

 POPUP "&Второй пункт"
 {
 MENUITEM "Трети&й", 3
 MENUITEM "Четверт&ый",4
 MENUITEM SEPARATOR
 POPUP "Еще подмен&ю"
 {
 MENUITEM "Десятый пунк&т", 6
 }
```

```
 }
 MENUITEM "Вы&ход", 5
}

//идентификаторы
#define IDI_ICON1 100

//определили пиктограмму
IDI_ICON1 ICON "ico1.ico"

//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 120
STYLE WS_POPUP | st
CAPTION "Пример немодального диалогового окна"
FONT 8, "Arial"
{
}
MENUP ACCELERATORS
{
 VK_F5, 4, VIRTKEY
}
;файл menu1.inc
;константы
;сообщение приходит при закрытии окна
WM_CLOSE        equ 10h
WM_INITDIALOG   equ 110h
WM_SETICON      equ 80h
WM_COMMAND      equ 111h
;прототипы внешних процедур
EXTERN  ShowWindow@8:NEAR
EXTERN  MessageBoxA@16:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  LoadMenuA@8:NEAR
EXTERN  SendMessageA@16:NEAR
EXTERN  SetMenu@8:NEAR
EXTERN  LoadAcceleratorsA@8:NEAR
EXTERN  TranslateAcceleratorA@12:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  CreateDialogParamA@20:NEAR
EXTERN  DestroyWindow@4:NEAR
EXTERN  TranslateMessage@4:NEAR
;структуры
```

```
;структура сообщения
MSGSTRUCT STRUC
      MSHWND    DD ?
      MSMESSAGE DD ?
      MSWPARAM  DD ?
      MSLPARAM  DD ?
      MSTIME    DD ?
      MSPT      DD ?
MSGSTRUCT ENDS
;файл menu1.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include menu1.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;сегмент данных
_DATA SEGMENT
      NEWHWND  DD 0
      MSG      MSGSTRUCT <?>
      HINST    DD 0 ;дескриптор приложения
      PA       DB "DIAL1",0
      PMENU    DB "MENUP",0
      STR1     DB "Выход из программы",0
      STR2     DB "Сообщение",0
      STR3     DB "Выбран четвертый",0
      ACC      DWORD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
        PUSH  0
        CALL  GetModuleHandleA@4
        MOV   HINST, EAX
;загрузить акселераторы
        PUSH  OFFSET PMENU
        PUSH  HINST
        CALL  LoadAcceleratorsA@8
        MOV   ACC,EAX   ; запомнить дескриптор таблицы
;создать немодальный диалог
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
```

```
      PUSH  [HINST]
      CALL  CreateDialogParamA@20
;визуализировать немодальный диалог
MOV  NEWHWND,EAX
      PUSH    1 ;SW_SHOWNORMAL
      PUSH    [NEWHWND]
      CALL    ShowWindow@8 ;показать созданное окно
;кольцо обработки сообщений
MSG_LOOP:
      PUSH    0
      PUSH    0
      PUSH    0
      PUSH    OFFSET MSG
      CALL    GetMessageA@16
      CMP     EAX, 0
      JE      END_LOOP
;транслировать сообщение акселератора
      PUSH    OFFSET MSG
      PUSH    ACC
      PUSH    NEWHWND
      CALL    TranslateAcceleratorA@12
      CMP     EAX,0
      JNE     MSG_LOOP
      PUSH    OFFSET MSG
      CALL    TranslateMessage@4
      PUSH    OFFSET MSG
      CALL    DispatchMessageA@4
      JMP     MSG_LOOP
END_LOOP:
      PUSH  0
      CALL  ExitProcess@4
;процедура окна
;расположение параметров в стеке
; [EBP+014H] LPARAM
; [EBP+10H]  WAPARAM
; [EBP+0CH]  MES
; [EBP+8]    HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;----------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
```

```
      JNE  L1
;закрыть диалоговое окно
      JMP L5
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  L3
;загрузить пиктограмму
      PUSH  100    ; идентификатор пиктограммы
      PUSH  HINST  ; идентификатор процесса
      CALL  LoadIconA@8
;установить пиктограмму
      PUSH  EAX
      PUSH  0      ; тип пиктограммы (маленькая)
      PUSH  WM_SETICON
      PUSH  DWORD PTR [EBP+08H]
      CALL  SendMessageA@16
;загрузить меню
      PUSH  OFFSET PMENU
      PUSH  HINST
      CALL  LoadMenuA@8
;установить меню
      PUSH  EAX
      PUSH  DWORD PTR [EBP+08H]
      CALL  SetMenu@8
;-------------------------------
      MOV   EAX,1  ; возвратить ненулевое значение
      JMP   FIN
;проверяем, не случилось ли чего с управляющими
;элементами на диалоговом окне
L3:
      CMP  DWORD PTR [EBP+0CH],WM_COMMAND
      JE   L6
      JMP  FINISH
;здесь определяем идентификатор, в данном случае
;это идентификатор пункта меню сообщения
L6:
      CMP  WORD PTR [EBP+10H],4
      JNE  L4
      PUSH 0         ;MB_OK
      PUSH OFFSET STR2
      PUSH OFFSET STR3
      PUSH 0
      CALL MessageBoxA@16
      JMP  FINISH
L4:
      CMP  WORD PTR [EBP+10H],5
      JNE  FINISH
```

```
;сообщение
      PUSH 0         ;MB_OK
      PUSH OFFSET STR2
      PUSH OFFSET STR1
      PUSH 0
      CALL MessageBoxA@16
;закрыть диалоговое немодальное окно
L5:
      PUSH DWORD PTR [EBP+08H]
      CALL DestroyWindow@4
;послать сообщение для выхода из кольца
;обработки сообщений
      PUSH  0
      CALL  PostQuitMessage@4 ;сообщение WM_QUIT
FINISH:
      MOV EAX,0
FIN:
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      RET 16
WNDPROC  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.4.3:

```
ml /c /coff menu1.asm
rc menu1.rc
link /subsystem:windows menu1.obj menu1.res
```

Несколько комментариев по поводу программы из листинга 2.4.3.

Немодальный диалог задается в файле ресурсов так же, как и модальный. Поскольку в свойствах окна нами не было указано свойство `WS_VISIBLE`, в программе, для того чтобы окно было видимым, мы используем функцию `ShowWindow`.

Выход из программы в нашем случае предполагает не только удаление из памяти диалогового окна, что достигается посредством функции `DestroyWindow`, но и выход из цикла обработки сообщений. Последнее осуществляется через вызов функции `PostQuitMessage`.

В заключение отмечу, что остались неосвещенными также следующие виды ресурсов.

❒ Ресурс, содержащий неструктурированные данные.

```
имя RCDATA
BEGIN
```

```
  raw-data
  . . .
END
```

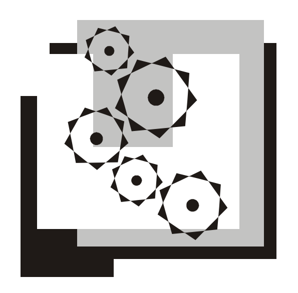
- Ресурс `VERSIONINFO`, предназначенный для хранения версий. Кроме версии исполняемого модуля здесь можно хранить оригинальное имя файла и требуемую операционную систему.

```
ID VERSIONINFO
BEGIN
  block-statement
  . . .
END
```

- Ресурс `FONT` — шрифт. Задание шрифта, как ресурса, осуществляется аналогично, например, курсору или пиктограмме.

```
#define ID_FONT FONT 1
ID_FONT FONT <имя_файла>
```

Эти ресурсы используются значительно реже остальных, и мы на них останавливаться не будем.



Глава 2.5

Примеры программ, использующих ресурсы

Основой обучения программированию является разбор конкретных работающих примеров, которые могут развиваться и дополняться самими обучаемыми. Нельзя учиться программированию на основе некоторых теоретических положений. К счастью, это не математика, и здесь аксиомы и теоремы не помогут. Программирование ближе к искусству. Это способ самовыражения. Я бы мог еще долго на эту тему рассуждать, да пора и делом заняться.

Вопрос использования ресурсов при программировании в Windows весьма важен, поэтому я посвящаю ему еще одну главу. Здесь будут приведены три более сложные программы на использование ресурсов и подробное их разъяснение.

Динамическое меню

Читатель, наверное, обращал внимание, что во многих программах меню может динамически меняться во время работы: исчезают и добавляются некоторые пункты, одно меню встраивается в другое и т. п. Пример простейших манипуляций с меню приведен в листинге 2.5.1.

Программа открывает окно с кнопкой и меню. При нажатии кнопки текущее меню заменяется другим. Если нажать еще раз, то меню исчезает. Следующее нажатие приводит к появлению первого меню и т. д., по кругу. Кроме того, в первом меню имеется пункт, который приводит к такому же результату, что и нажатие кнопки. Наконец, для этого пункта установлена акселераторная клавиша — <F5>. При передвижении по меню названия пунктов меню и заголовков выпадающих (popup) подменю отображаются в заголовке окна. Вот, вкратце, как работает программа. Механизмы работы программы будут подробно разобраны далее.

Листинг 2.5.1. Пример манипуляции меню

```
//файл menu2.rc
//виртуальная клавиша <F5>
#define VK_F5   0x74

//********* MENUP ***********

MENUP MENU
{
 POPUP "&Первый пункт"
 {
 MENUITEM "&Первый", 1
 MENUITEM "В&торой", 2
 }

 POPUP "&Второй пункт"
 {
 MENUITEM "Трети&й", 3
 MENUITEM "Четверт&ый",4
 MENUITEM SEPARATOR
 POPUP "Еще подмен&ю"
 {
 MENUITEM "Дополнительный пу&нкт", 6
 }


 }

 MENUITEM "Вы&ход", 5
}
//********* MENUC ***********
MENUC MENU
{
 POPUP "Набор первый"
 {
 MENUITEM "Белый", 101
 MENUITEM "Серый", 102
 MENUITEM "Черный", 103
 }
 POPUP "Набор второй"
 {
 MENUITEM "Красный", 104
 MENUITEM "Синий", 105
 MENUITEM "Зеленый", 106
 }
```

```
}

//таблица акселераторов
//определен один акселератор для вызова пункта из меню MENUP
MENUP ACCELERATORS
{
 VK_F5, 4, VIRTKEY, NOINVERT
}

;файл menu.inc
;константы
;сообщение приходит при закрытии окна
WM_DESTROY          equ 2
;сообщение приходит при создании окна
WM_CREATE           equ 1
;сообщение при щелчке левой кнопкой мыши в области окна
WM_SYSCOMMAND       equ 112h
WM_COMMAND          equ 111h
WM_MENUSELECT       equ 11Fh
WM_SETTEXT          equ 0Ch
MIIM_STRING         equ 40h
MF_STRING           equ 0h
MF_POPUP            equ 10h
;свойства окна
CS_VREDRAW          equ 1h
CS_HREDRAW          equ 2h
CS_GLOBALCLASS      equ 4000h
WS_OVERLAPPEDWINDOW  equ 000CF0000H
STYLE equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
BS_DEFPUSHBUTTON    equ 1h
WS_VISIBLE          equ 10000000h
WS_CHILD            equ 40000000h
STYLBTN equ WS_CHILD+BS_DEFPUSHBUTTON+WS_VISIBLE
;идентификатор стандартной пиктограммы
IDI_APPLICATION     equ 32512
;идентификатор курсора
IDC_ARROW           equ 32512
;режим показа окна - нормальный
SW_SHOWNORMAL       equ 1
SW_HIDE             equ 0
SW_SHOWMINIMIZED    equ 2
;прототипы внешних процедур
EXTERN  wsprintfA:NEAR
EXTERN  GetMenuItemInfoA@16:NEAR
EXTERN  LoadMenuA@8:NEAR
EXTERN  SendMessageA@16:NEAR
```

```
EXTERN  MessageBoxA@16:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
EXTERN  TranslateAcceleratorA@12:NEAR
EXTERN  LoadAcceleratorsA@8:NEAR
EXTERN  GetMenu@4:NEAR
EXTERN  DestroyMenu@4:NEAR
EXTERN  SetMenu@8:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;----структура класса окон
WNDCLASS STRUC
      CLSSTYLE   DD ?
      CLWNDPROC  DD ?
      CLSCBCLSEX DD ?
      CLSCBWNDEX DD ?
      CLSHINST   DD ?
      CLSHICON   DD ?
      CLSHCURSOR DD ?
      CLBKGROUND DD ?
      CLMENNAME  DD ?
      CLNAME     DD ?
WNDCLASS ENDS
MENINFO STRUCT
      cbSize        DD ?
      fMask         DD ?
      fType         DD ?
```

```
      fState        DD ?
      wID           DD ?
      hSubMenu      DD ?
      hbmpChecked   DD ?
      hbmpUnchecked DD ?
      dwItemData    DD ?
      dwTypeData    DD ?
      cch           DD ?
MENINFO ENDS

;файл menu.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include menu.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      SPACE      DB 30 dup(32),0
      MENI       MENINFO   <0>
      NEWHWND    DD 0
      MSG        MSGSTRUCT <?>
      WC         WNDCLASS  <?>
      HINST      DD 0             ; дескриптор приложения
      CLASSNAME  DB 'CLASS32',0
      CPBUT      DB 'Кнопка',0    ; выход
      CLSBUTN    DB 'BUTTON',0
      HWNDBTN    DD 0
      CAP        DB 'Сообщение',0
      MES        DB 'Конец работы программы',0
      MEN        DB 'MENUP',0
      MENC       DB 'MENUC',0
      ACC        DD ?
      HMENU      DD ?
      PRIZN      DD ?
      BUFER      DB 100 DUP(0)
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;инициализировать счетчик
      MOV    PRIZN,2
```

```
;получить дескриптор приложения
      PUSH   0
      CALL   GetModuleHandleA@4
      MOV    HINST, EAX
REG_CLASS:
;заполнить структуру окна
;стиль
      MOV    WC.CLSSTYLE,STYLE
;процедура обработки сообщений
      MOV    WC.CLWNDPROC, OFFSET WNDPROC
      MOV    WC.CLSCBCLSEX, 0
      MOV    WC.CLSCBWNDEX, 0
      MOV    EAX, HINST
      MOV    WC.CLSHINST, EAX
;пиктограмма окна
      PUSH   IDI_APPLICATION
      PUSH   0
      CALL   LoadIconA@8
      MOV    WC.CLSHICON, EAX
;курсор окна
      PUSH   IDC_ARROW
      PUSH   0
      CALL   LoadCursorA@8
      MOV    WC.CLSHCURSOR, EAX
;----------
      MOV    WC.CLBKGROUND, 17   ; цвет окна
      MOV    DWORD PTR WC.CLMENNAME,OFFSET MEN
      MOV    DWORD PTR WC.CLNAME,OFFSET CLASSNAME
      PUSH   OFFSET WC
      CALL   RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH   0
      PUSH   HINST
      PUSH   0
      PUSH   0
      PUSH   400      ; DY - высота окна
      PUSH   400      ; DX - ширина окна
      PUSH   100      ; Y
      PUSH   100      ; X
      PUSH   WS_OVERLAPPEDWINDOW
      PUSH   OFFSET SPACE       ; имя окна
      PUSH   OFFSET CLASSNAME   ; имя класса
      PUSH   0
      CALL   CreateWindowExA@48
;проверка на ошибку
      CMP    EAX,0
```

```
      JZ    _ERR
      MOV    NEWHWND, EAX ;дескриптор окна
;определить идентификатор меню
      PUSH   EAX
      CALL   GetMenu@4
      MOV    HMENU,EAX
;загрузить акселераторы
      PUSH   OFFSET MEN
      PUSH   HINST
      CALL   LoadAcceleratorsA@8
      MOV    ACC,EAX
;-----------------------------------
      PUSH   SW_SHOWNORMAL
      PUSH   NEWHWND
      CALL   ShowWindow@8 ;показать созданное окно
;-----------------------------------
      PUSH   NEWHWND
      CALL   UpdateWindow@4 ;команда перерисовать видимую
                            ;часть окна, сообщение WM_PAINT
;цикл обработки сообщений
MSG_LOOP:
      PUSH   0
      PUSH   0
      PUSH   0
      PUSH   OFFSET MSG
      CALL   GetMessageA@16
      CMP    EAX, 0
      JE     END_LOOP
      PUSH   OFFSET MSG
      PUSH   ACC
      PUSH   NEWHWND
      CALL   TranslateAcceleratorA@12
      CMP    EAX,0
      JNE    MSG_LOOP
      PUSH   OFFSET MSG
      CALL   TranslateMessage@4
      PUSH   OFFSET MSG
      CALL   DispatchMessageA@4
      JMP    MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
      PUSH   MSG.MSWPARAM
      CALL   ExitProcess@4
_ERR:
      JMP    END_LOOP
;---------------------------------
```

```
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC PROC
      PUSH  EBP
      MOV   EBP,ESP
      PUSH  EBX
      PUSH  ESI
      PUSH  EDI
      MOV   EAX,HWNDBTN
;сообщение WM_DESTROY - при закрытии окна
      CMP   DWORD PTR [EBP+0CH], WM_DESTROY
      JE    WMDESTROY
;сообщение WM_CREATE - при создании окна
      CMP   DWORD PTR [EBP+0CH], WM_CREATE
      JE    WMCREATE
;сообщение WM_COMMAND - при событиях
;с элементами на окне
      CMP   DWORD PTR [EBP+0CH], WM_COMMAND
      JE    WMCOMMND
;сообщение WM_MENUSELECT - события, связанные с меню
      CMP   DWORD PTR [EBP+0CH], WM_MENUSELECT
      JE    WMMENUSELECT
;остальные события возвращаем обратно
      JMP   DEFWNDPROC
WMMENUSELECT:
;проверяем, что активизировано - пункт меню
;или заголовок выпадающего меню
      XOR   EDX,EDX
      TEST  WORD PTR [EBP+12H],MF_POPUP
      SETNE DL
;заполнение структуры для вызова функции
;GetMenuItemInfo
      MOVZX EAX,WORD PTR [EBP+10H] ;идентификатор
      MOV   MENI.cbSize,48
      MOV   MENI.fMask,MIIM_STRING
      MOV   MENI.fType,MF_STRING
      MOV   EBX,DWORD PTR [EBP+14H]
      MOV   MENI.hSubMenu,EBX
      MOV   MENI.dwTypeData,OFFSET BUFER
      MOV   MENI.cch,100
;получить информацию о выбранном пункте меню
      PUSH  OFFSET MENI
```

```
      PUSH  EDX
      PUSH  EAX
      PUSH  DWORD PTR [EBP+14H]
      CALL  GetMenuItemInfoA@16
;проверить результат выполнения функции
      CMP   EAX,0
      JE    FINISH
;вывести название пункта меню
      PUSH  MENI.dwTypeData
      PUSH  0
      PUSH  WM_SETTEXT
      PUSH  DWORD PTR [EBP+08H]
      CALL  SendMessageA@16
      MOV   EAX,0
      JMP   FINISH
WMCOMMND:
;проверить, не нажата ли кнопка
      CMP   DWORD PTR [EBP+14H], EAX
      JE    YES_BUT
;проверить, не выбран ли пункт меню MENUC - Выход
      CMP   WORD PTR [EBP+10H],5
      JE    WMDESTROY
;проверить, не выбран ли пункт меню с идентификатором 4
      CMP   WORD PTR [EBP+10H],4
      JE    YES_BUT
      JMP   DEFWNDPROC
YES_BUT:
;здесь обработка нажатия кнопки мыши
;вначале стереть надпись в заголовке
      PUSH   OFFSET SPACE
      PUSH   0
      PUSH   WM_SETTEXT
      PUSH   DWORD PTR [EBP+08H]
      CALL   SendMessageA@16
;проверить, загружено или нет меню
      PUSH   HMENU
      CALL   DestroyMenu@4
      CMP    PRIZN,1
      JE     L2
;загрузить меню MENC
      PUSH   OFFSET MENC
      PUSH   HINST
      CALL   LoadMenuA@8
;установить меню
      MOV    HMENU,EAX
      PUSH   EAX
```

```
        PUSH    DWORD PTR [EBP+08H]
        CALL    SetMenu@8
;установить признак
        MOV     PRIZN,1
        MOV     EAX,0
        JMP     FINISH
L2:
;загрузить меню MENUP
        PUSH    OFFSET MEN
        PUSH    HINST
        CALL    LoadMenuA@8
;установить меню
        MOV     HMENU,EAX
        PUSH    EAX
        PUSH    DWORD PTR [EBP+08H]
        CALL    SetMenu@8
;установить признак
        MOV     PRIZN,2
        MOV     EAX,0
        JMP     FINISH
WMCREATE:
;создать окно-кнопку
        PUSH    0
        PUSH    HINST
        PUSH    0
        PUSH    DWORD PTR [EBP+08H]
        PUSH    20      ; DY
        PUSH    60      ; DX
        PUSH    10      ; Y
        PUSH    10      ; X
        PUSH    STYLBTN
;имя окна (надпись на кнопке)
        PUSH    OFFSET CPBUT
        PUSH    OFFSET CLSBUTN ;имя класса
        PUSH    0
        CALL    CreateWindowExA@48
        MOV     HWNDBTN,EAX ;запомнить дескриптор кнопки
        MOV     EAX, 0
        JMP     FINISH
DEFWNDPROC:
        PUSH    DWORD PTR [EBP+14H]
        PUSH    DWORD PTR [EBP+10H]
        PUSH    DWORD PTR [EBP+0CH]
        PUSH    DWORD PTR [EBP+08H]
        CALL    DefWindowProcA@16
        JMP     FINISH
```

```
WMDESTROY:
      PUSH   0         ;MB_OK
      PUSH   OFFSET CAP
      PUSH   OFFSET MES
      PUSH   DWORD PTR [EBP+08H] ;дескриптор окна
      CALL   MessageBoxA@16
      PUSH   0
      CALL   PostQuitMessage@4 ;сообщение WM_QUIT
      MOV    EAX, 0
FINISH:
      POP    EDI
      POP    ESI
      POP    EBX
      POP    EBP
      RET    16
WNDPROC ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.5.1:

```
ml /c /coff menu.asm
rc menu.rc
link /subsystem:windows menu.obj menu.res
```

Программа в листинге 2.5.1 имеет ряд механизмов, к обсуждению которых я намерен сейчас приступить. Для начала замечу, что в программе используются три ресурса: два меню и таблица акселераторов (см. файл ресурсов в листинге 2.5.1).

Первое, на что хочу обратить ваше внимание, — это переменная `PRIZN`. В ней хранится состояние меню: значение `2` — загружено меню `MENUP`, значение `1` — загружено `MENUC`. Начальное состояние обеспечивается заданием меню при регистрации класса окна:

```
MOV   DWORD PTR [WC.CLMENNAME],OFFSET MEN
```

Второе — это кнопка. Механизм распознавания нажатия кнопки мы уже разбирали, так что больше на этом останавливаться не будем. Событие, которое происходит при нажатии кнопки — смена меню.

Выбор одного из пунктов меню (пункт **Четвертый**) с именем `MENUP` также приводит к смене меню. Здесь должно быть все понятно, поскольку обращение идет к тому же участку программы, что и в случае нажатия кнопки.

Интересная ситуация возникает с акселератором. Акселераторная клавиша у нас <F5>. При ее нажатии генерируется такое же сообщение, как при выборе пункта **Четвертый** меню `MENUP`. Важно то, что такое же сообщение будет

генерироваться и тогда, когда загружается меню с именем MENUC и когда меню не будет. А поскольку наша процедура обрабатывает сообщение в любом случае, клавиша <F5> будет срабатывать всегда.

Рассмотрим теперь, как производится определение названия выбранного пункта меню. Центральную роль в этом механизме играет сообщение WM_MENUSELECT. Это сообщение приходит всегда, когда выбирается пункт меню. После получения сообщения WM_MENUSELECT в младшем слове параметра WPARAM может содержаться либо идентификатор пункта меню, либо номер заголовка выпадающего меню. Это ключевой момент. Нам важно это знать, т. к. строка заголовка выпадающего меню и строка пункта меню получаются по-разному. Определить, что выбрано, можно по старшему слову параметра WPARAM. Мы используем для этого константу MF_POPUP: TEST WORD PTR [EBP+12H],MF_POPUP. Обратите внимание, как удобна и как кстати здесь команда SETNE.

Далее, для получения строки-названия пункта используется функция GetMenuItemInfo. Третьим параметром этой функции как раз и может быть либо ноль, либо единица. Если ноль, то второй параметр — это идентификатор пункта меню, если единица, то второй параметр — номер заголовка выпадающего меню. Четвертым параметром является указатель на структуру, которая и будет заполняться в результате выполнения функции. Некоторые поля этой структуры должны быть, однако, заполнены заранее. Обращаю внимание на поле dwTypeData, которое должно содержать указатель на буфер, получающий необходимую нам строку. При этом поле cch должно содержать длину этого буфера. Но для того чтобы поля dwTypeData и cch трактовались функцией именно как указатель на буфер и его длину, поля fMask и fType должны быть правильно заполнены (см. листинг 2.5.1). Наконец, поле cbSize должно содержать длину всей структуры.

После получения нужной информации, т. е. строки-названия пункта меню, при помощи функции SendMessage мы посылаем сообщение WM_SETTEXT, которое дает команду установить заголовок окна. В заголовке окна, таким образом, будет установлено название пункта меню.

Горячие клавиши

Итак, продолжим рассматривать ресурсы. Хочется рассказать о весьма интересном приеме, который можно использовать при работе с окнами редактирования. Наверное, вы работали с визуальными языками типа Visual Basic, Delphi и пр. и обратили внимание, что поля редактирования можно так запрограммировать, а точнее, задать их свойства, что они позволят вводить только вполне определенные символы. В Delphi это свойство называется EditMask.

Я думаю, вам хотелось бы понять, как подобное реализовать только API-средствами. Но обо всем по порядку.

Обычное окно при нажатии клавиши (если оно активно) получает сообщения WM_KEYDOWN, WM_KEYUP и их квинтэссенцию WM_CHAR. Но в данном случае мы имеем дело не с обычным окном, а с диалоговым. Диалоговое окно (модальное) таких сообщений не получает. Остается надеяться на сообщения, посылаемые в ответ на события, происходящие с самим элементом — "окном редактирования". Но, увы, и здесь нас ждут разочарования. Данный элемент получает лишь два сообщения из тех, которые нас хоть как-то могут заинтересовать. Это сообщение EN_UPDATE и сообщение EN_CHANGE. Оба сообщения приходят, когда уже произведено изменение в окне редактирования. Но сообщение EN_UPDATE приходит, когда изменения на экране еще не произведены, а EN_CHANGE — после таких изменений. Нам придется сначала получить содержимое окна редактирования, определить, какой символ туда поступил последним, и если он недопустим, удалить его из строки и послать строку в окно снова. Добавьте сюда еще проблему, связанную с положением курсора и вторичным приходом сообщения EN_UPDATE. Лично я по такому пути бы не пошел.

Есть другой более изящный и короткий путь: использовать понятие *горячей клавиши* (hotkey). Мы ограничимся лишь программными свойствами горячих клавиш, т. е. свойствами, которые необходимо знать программисту, чтобы использовать горячие клавиши в своих программах.

Горячая клавиша может быть связана с любой виртуальной клавишей, клавишей, определяемой через макроконстанты с префиксом VK. Для обычных алфавитно-цифровых клавиш значение этих констант просто совпадает с кодами ASCII. Возможны также сочетания с управляющими клавишами <Alt>, <Ctrl>, <Shift>. После того как для данного окна определена горячая клавиша, при ее нажатии в функцию окна приходит сообщение WM_HOTKEY. По параметрам можно определить, какая именно горячая клавиша была нажата. Существенно, что понятие горячей клавиши глобально, т. е. она будет срабатывать, если будут активны другие окна и даже окна других приложений. Это требует от программиста весьма аккуратной работы, т. к. вы можете заблокировать нормальную работу других приложений. То есть необходимо отслеживать, когда данное окно активно, а когда нет. Этому могут помочь сообщения WM_ACTIVATE и WM_ACTIVATEAPP. Первое сообщение всегда приходит в функцию окна тогда, когда окно активизируется или деактивизируется. Первый раз сообщение приходит при создании окна. Вот при получении этого сообщения и есть резон зарегистрировать горячие клавиши. Второе сообщение всегда приходит в функцию окна, когда окно теряет фокус — активи-

зируется другое окно. Соответственно, при получении данного сообщения и следует отменить регистрацию этих клавиш.

Для работы с горячими клавишами используют в основном две функции: `RegisterHotKey` и `UnregisterHotKey`. Функция `RegisterHotKey` имеет следующие параметры:

- 1-й параметр — дескриптор окна;
- 2-й параметр — идентификатор горячей клавиши;
- 3-й параметр — модификатор, определяющий, не нажата ли управляющая клавиша;
- 4-й параметр — виртуальный код клавиши.

Функция `UnregisterHotKey` имеет всего два параметра:

- 1-й параметр — дескриптор окна;
- 2-й параметр — идентификатор.

Важно, что если мы определили горячую клавишу, она перестает участвовать в каких-либо событиях, фактически оказывается заблокированной. Единственный метод, с помощью которого можно судить о нажатии этой клавиши, — сообщение `WM_HOTKEY`.

Рассмотрим простой пример диалогового окна, на котором расположены два поля редактирования и кнопка выхода. Поставим перед собой такую цель. Первое поле редактирования должно пропускать только цифры от 0 до 9. Во второе поле можно вводить все символы. Ранее рассматривался возможный механизм использования горячих клавиш с сообщениями `WM_ACTIVATE` и `WM_ ACTIVATEAPP`. Ясно, что эти события в данном случае нам ничем не помогут. Здесь задача тоньше, надо использовать сообщения, относящиеся к одному полю редактирования. Это сообщения `EN_SETFOCUS` и `EN_KILLFOCUS`, передаваемые, естественно, через сообщение `WM_COMMAND`. Далее представлена программа, демонстрирующая этот механизм, и комментарий к ней. Сообщение `EN_SETFOCUS` говорит о том, что окно редактирования приобрело фокус (стало активным), а сообщение `EN_KILLFOCUS` — что окно редактирования потеряло фокус. В листинге 2.5.2 представлена программа использования горячих клавиш.

Листинг 2.5.2. Пример использования горячих клавиш с диалоговым окном

```
//файл dial1.rc
//определение констант
//стили окна
#define WS_SYSMENU      0x00080000L
```

```
#define WS_MINIMIZEBOX  0x00020000L
#define WS_MAXIMIZEBOX  0x00010000L
//текст в окне редактирования прижат к левому краю
#define ES_LEFT         0x0000L
//стиль всех элементов в окне
#define WS_CHILD        0x40000000L
//элементы в окне должны быть изначально видимы
#define WS_VISIBLE      0x10000000L
//бордюр вокруг элемента
#define WS_BORDER       0x00800000L
//при помощи клавиши <Tab> можно по очереди активизировать элементы
#define WS_TABSTOP      0x00010000L
//прижать строку к левому краю отведенного поля
#define SS_LEFT         0x00000000L
//стиль "кнопка"
#define BS_PUSHBUTTON   0x00000000L
//центрировать текст на кнопке
#define BS_CENTER       0x00000300L
#define DS_LOCALEDIT    0x20L

//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 120
STYLE WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
CAPTION "Пример диалогового окна"
FONT 8, "Arial"
{
//поле редактирования, идентификатор 1
 CONTROL "", 1, "edit", ES_LEFT | WS_CHILD
 | WS_VISIBLE | WS_BORDER | WS_TABSTOP, 24, 20, 128, 12
//еще одно поле редактирования, идентификатор 2
 CONTROL "", 2, "edit", ES_LEFT | WS_CHILD
 | WS_VISIBLE | WS_BORDER | WS_TABSTOP, 24, 52, 127, 12
//текст, идентификатор 3
 CONTROL "Строка 1", 3, "static", SS_LEFT
 | WS_CHILD | WS_VISIBLE, 164, 22, 60, 8
//еще текст, идентификатор 4
 CONTROL "Строка 2", 4, "static", SS_LEFT
 | WS_CHILD | WS_VISIBLE, 163, 54, 60, 8
//кнопка, идентификатор 5
 CONTROL "Выход", 5, "button", BS_PUSHBUTTON
 | BS_CENTER | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
 180, 76, 50, 14
}
;файл dial1.inc
;константы
;сообщение приходит при закрытии окна
```

```
WM_CLOSE            equ 10h
WM_INITDIALOG       equ 110h
WM_COMMAND          equ 111h
WM_SETTEXT          equ 0Ch
WM_HOTKEY           equ 312h
EN_SETFOCUS         equ 100h
EN_KILLFOCUS        equ 200h
;прототипы внешних процедур
EXTERN  UnregisterHotKey@8:NEAR
EXTERN  RegisterHotKey@16:NEAR
EXTERN  MessageBoxA@16:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  SendMessageA@16:NEAR
EXTERN  GetDlgItem@8:NEAR
EXTERN  MessageBoxA@16:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DWORD ?
      MSMESSAGE  DWORD ?
      MSWPARAM   DWORD ?
      MSLPARAM   DWORD ?
      MSTIME     DWORD ?
      MSPT       DWORD ?
MSGSTRUCT ENDS
;файл dial.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include dial1.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
MSG MSGSTRUCT <?>
HINST DD 0    ;дескриптор приложения
PA  DB "DIAL1",0
STR1 DB "Неправильный символ!",0
STR2 DB "Ошибка!",0
;таблица для создания горячих клавиш
```

```
TAB      DB 32,33,34,35,36,37,38,39,40
         DB 41, 42,43,44,45,46,47,58,59,60
         DB 61, 62,63,64,65,66,67,68,69,70
         DB 71, 72,73,74,75,76,77,78,79,80
         DB 81, 82,83,84,85,86,87,88,89,90
         DB 91, 92,93,94,95,96,97,98,99,100
         DB 101,102,103,104,105,106,107,108,109,110
         DB 111,112,113,114,115,116,117,118,119,120
         DB 121,122,123,124,125,126,127,128,129,130
         DB 131,132,133,134,135,136,137,138,139,140
         DB 141,142,143,144,145,146,147,148,149,150
         DB 151,152,153,154,155,156,157,158,159,160
         DB 161,162,163,164,165,166,167,168,169,170
         DB 171,172,173,174,175,176,177,178,179,180
         DB 181,182,183,184,185,186,187,188,189,190
         DB 191,192,193,194,195,196,197,198,199,200
         DB 201,202,203,204,205,206,207,208,209,210
         DB 211,212,213,214,215,216,217,218,219,220
         DB 221,222,223,224,225,226,227,228,229,230
         DB 231,232,233,234,235,236,237,238,239,240
         DB 241,242,243,244,245,246,247,248,249,250
         DB 251,252,253,254,255
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   [HINST], EAX
;---------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  [HINST]
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
KOL:
;---------------------------------
      PUSH  0
      CALL  ExitProcess@4
;---------------------------------
;процедура окна
;расположение параметров в стеке
```

```
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC PROC
      PUSH  EBP
      MOV   EBP,ESP
      PUSH  EBX
      PUSH  ESI
      PUSH  EDI
;--------------------------------
      CMP   DWORD PTR [EBP+0CH],WM_CLOSE
      JNE   L1
      PUSH  0
      PUSH  DWORD PTR [EBP+08H]
      CALL  EndDialog@8
      MOV   EAX,1
      JMP   FIN
L1:
      CMP   DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE   L2
;здесь заполнить окна редактирования, если надо
;
      MOV   EAX,1
      JMP   FIN
L2:
      CMP   DWORD PTR [EBP+0CH],WM_COMMAND
      JNE   L5
;кнопка выхода?
      CMP   WORD PTR [EBP+10H],5
      JNE   L3
      PUSH  0
      PUSH  DWORD PTR [EBP+08H]
      CALL  EndDialog@8
      MOV   EAX,1
      JMP   FIN
L3:
      CMP   WORD PTR [EBP+10H],1
      JNE   FINISH
;блок обработки сообщений первого окна редактирования
      CMP   WORD PTR [EBP+12H],EN_KILLFOCUS
      JNE   L4
;окно редактирования с идентификатором 1 теряет фокус
      MOV   EBX,0
;снимаем все горячие клавиши
```

```
L33:
      MOVZX EAX,BYTE PTR [TAB+EBX]
      PUSH  EAX
      PUSH  DWORD PTR [EBP+08H]
      CALL  UnregisterHotKey@8
      INC   EBX
      CMP   EBX,214
      JNE   L33
      MOV   EAX,1
      JMP   FIN
L4:
      CMP   WORD PTR [EBP+12H],EN_SETFOCUS
      JNE   FINISH
;окно редактирования с идентификатором 1 получает фокус
      MOV   EBX,0
;устанавливаем горячие клавиши
L44:
      MOVZX EAX,BYTE PTR [TAB+EBX]
      PUSH  EAX
      PUSH  0
      PUSH  EAX
      PUSH  DWORD PTR [EBP+08H]
      CALL  RegisterHotKey@16
      INC   EBX
      CMP   EBX,214
      JNE   L44
      MOV   EAX,1
      JMP   FIN
L5:
      CMP   DWORD PTR [EBP+0CH],WM_HOTKEY
      JNE   FINISH
;здесь реакция на неправильно введенный символ
      PUSH  0         ;MB_OK
      PUSH  OFFSET STR2
      PUSH  OFFSET STR1
      PUSH  DWORD PTR [EBP+08H] ;дескриптор окна
      CALL  MessageBoxA@16
FINISH:
      MOV   EAX,0
FIN:
      POP   EDI
      POP   ESI
      POP   EBX
      POP   EBP
      RET   16
```

```
WNDPROC ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.5.2:

```
ml /c /coff dial.asm
rc dial.rc
link /subsystem:windows dial.obj dial.res
```

Прокомментирую программу из листинга 2.5.2.

Самое главное — разберитесь с тем, как мы определяем, когда первое поле редактирования теряет, когда приобретает фокус. Вначале определяется, что сообщение пришло от поля редактирования с идентификатором 1 (см. текст файла ресурсов), а затем — какое сообщение пришло: EN_SETFOCUS или EN_KILLFOCUS. В первом случае мы устанавливаем горячие клавиши, а во втором выключаем их.

В области данных задаем таблицу горячих клавиш. Функция RegisterHotKey имеет следующие параметры:

- 1-й параметр — идентификатор окна, куда должно прийти сообщение WM_HOTKEY. Если параметр равен 0, то сообщение WM_HOTKEY должно обрабатываться в цикле обработки сообщений;
- 2-й параметр — идентификатор горячей клавиши. В нашем приложении (см. листинг 2.5.2) в качестве идентификатора выбирается код клавиши;
- 3-й параметр — флаг нажатия управляющих клавиш. Можно использовать следующие константы: MOD_ALT = 1 (для клавиши <Alt>), MOD_CONTROL = 2 (для клавиш <Ctrl>), MOD_SHIFT = 4 (для клавиш <Shift>), MOD_WIN = 8 (для остальных специальных клавиш, обрабатываемых Windows). Для того чтобы не обрабатывать управляющие клавиши, значение параметра должно быть равно 0;
- 4-й параметр — виртуальный код клавиши. Как я уже неоднократно говорил, для алфавитно-цифровых клавиш этот код совпадает с кодом ASCII.

В приложении из листинга 2.5.2 мы создаем горячие клавиши для всех алфавитно-цифровых клавиш, кроме клавиш с виртуальными кодами 48—57, которые соответствуют цифрам 0—9. Тем самым отсекаем всю вводимую информацию кроме цифровой.

Функция UnregisterHotKey, которая используется для удаления горячих клавиш, имеет всего два параметра: дескриптор окна и идентификатор клавиши (см. листинг 2.5.2).

В нашем случае виртуальный код клавиши и идентификатор горячей клавиши совпадают. Это сделано просто для удобства. Конечно, здесь есть поле

для дальнейшего усовершенствования. Скажем, исключить из обработки клавиши управления курсором. Я думаю, читатель может справиться с этим самостоятельно.

Управление списками

В данном разделе будет рассмотрен пример программы с двумя списками (листинг 2.5.3). Двойным щелчком по элементу левого списка заполняется правый список. При этом мы учитываем возможность повторного щелчка по одному и тому же элементу. В принципе, в программе нет ничего сложного. Ниже будет дан комментарий к ней. Но мне хотелось бы немного поговорить о таком элементе, как список, остановившись на некоторых важных моментах.

Средства управления списком можно разделить на сообщения и свойства.[1] Свойства задаются в файле ресурсов. Например, свойство `LBS_SORT` приводит к тому, что содержимое списка будет автоматически сортироваться при добавлении туда элемента. Очень важным является свойство `LBS_WANTKEYBOARDINPUT`. При наличии такого свойства приложение получает сообщение `WM_VKEYTOITEM`, которое посылается приложению, когда нажимается какая-либо клавиша при наличии фокуса на данном списке. Вы можете выбрать самостоятельную обработку — клавиша <PgUp>, или оставить стандартную обработку. В том случае, если стандартная обработка не нужна, следует возвратить из функции диалогового окна отрицательное значение.

Листинг 2.5.3. Пример программы с двумя списками. Перебросить запись из левого списка в правый список можно двойным щелчком мыши или клавишей <Insert>

```
//файл diallst.rc
//определение констант
#define WS_SYSMENU      0x00080000L
#define WS_MINIMIZEBOX  0x00020000L
#define WS_MAXIMIZEBOX  0x00010000L
#define WS_VISIBLE      0x10000000L
#define WS_TABSTOP      0x00010000L
#define WS_VSCROLL      0x00200000L
#define WS_THICKFRAME   0x00040000L
```

[1] Впрочем, это можно сказать о любых элементах в диалоговом окне. Не правда ли, это весьма похоже на методы и свойства в объектном программировании. Но мы-то с вами знаем, что если углубиться еще дальше, то обнаружим, что значительная часть свойств опять сведется к обработке сообщений (см. комментарий к программе из листинга 2.5.3).

```
#define LBS_NOTIFY      0x0001L
#define LBS_SORT        0x0002L
#define LBS_WANTKEYBOARDINPUT 0x0400L

//идентификаторы
#define LIST1     101
#define LIST2     102
#define IDI_ICON1 3

//определили пиктограмму
IDI_ICON1 ICON "ico1.ico"

//определение диалогового окна
DIAL1 DIALOG 0, 0, 210, 110
STYLE WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
CAPTION "Пример диалогового окна"
FONT 8, "Arial"
{
 CONTROL "ListBox1",LIST1,"listbox", WS_VISIBLE |
 WS_TABSTOP | WS_VSCROLL | WS_THICKFRAME |
 LBS_NOTIFY|LBS_WANTKEYBOARDINPUT,
 16, 16, 70, 75
 CONTROL "ListBox2", LIST2, "listbox", WS_VISIBLE |
 WS_TABSTOP | WS_VSCROLL | WS_THICKFRAME |LBS_NOTIFY |
 LBS_SORT, 116, 16, 70, 75
}

;файл diallst.inc
;константы
;сообщение приходит при закрытии окна
WM_CLOSE        equ 10h
WM_INITDIALOG   equ 110h
WM_SETICON      equ 80h
WM_COMMAND      equ 111h
WM_VKEYTOITEM   equ 2Eh
LB_ADDSTRING    equ 180h
LBN_DBLCLK      equ 2
LB_GETCURSEL    equ 188h
LB_GETTEXT      equ 189h
LB_FINDSTRING   equ 18Fh
VK_INSERT       equ 2Dh

;прототипы внешних процедур
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
```

```
EXTERN  EndDialog@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  SendMessageA@16:NEAR
EXTERN  SendDlgItemMessageA@20:NEAR
EXTERN  MessageBoxA@16:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS

;файл diallst.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include diallst.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG      MSGSTRUCT <?>
      HINST    DD 0 ;дескриптор приложения
      PA       DB "DIAL1",0
      BUFER    DB 100 DUP(0)
      STR1     DB "Первый",0
      STR2     DB "Второй",0
      STR3     DB "Третий",0
      STR4     DB "Четвертый",0
      STR5     DB "Пятый",0
      STR6     DB "Шестой",0
      STR7     DB "Седьмой",0
      STR8     DB "Восьмой",0
      STR9     DB "Девятый",0
      STR10    DB "Десятый",0
      STR11    DB "Одиннадцатый",0
      STR12    DB "Двенадцатый",0
      STR13    DB "Тринадцатый",0
      STR14    DB "Четырнадцатый",0
      STR15    DB "Пятнадцатый",0
```

```
      INDEX    DD OFFSET STR1
               DD OFFSET STR2
               DD OFFSET STR3
               DD OFFSET STR4
               DD OFFSET STR5
               DD OFFSET STR6
               DD OFFSET STR7
               DD OFFSET STR8
               DD OFFSET STR9
               DD OFFSET STR10
               DD OFFSET STR11
               DD OFFSET STR12
               DD OFFSET STR13
               DD OFFSET STR14
               DD OFFSET STR15
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV  [HINST], EAX
;-------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  [HINST]
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
;сообщение об ошибке
KOL:
;--------------------------------
      PUSH  0
      CALL  ExitProcess@4
;--------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC PROC
      PUSH EBP
```

```
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;--------------------------------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP  FINISH
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  L2
;загрузить пиктограмму
      PUSH  3         ; идентификатор пиктограммы
      PUSH [HINST]    ; идентификатор процесса
CALL LoadIconA@8
;установить пиктограмму
      PUSH  EAX
      PUSH  0         ; тип пиктограммы (маленькая)
      PUSH  WM_SETICON
      PUSH  DWORD PTR [EBP+08H]
      CALL  SendMessageA@16
;заполнить левый список
      MOV   ECX,15
      MOV   ESI,0
LO1:
      PUSH  ECX       ; сохранить параметр цикла
      PUSH  INDEX[ESI]
      PUSH  0
      PUSH  LB_ADDSTRING
      PUSH  101
      PUSH  DWORD PTR [EBP+08H]
      CALL  SendDlgItemMessageA@20
      ADD   ESI,4
      POP   ECX
      LOOP  LO1
      JMP   FINISH
L2:
      CMP   DWORD PTR [EBP+0CH],WM_COMMAND
      JNE   L3
;не сообщение ли от левого списка?
      CMP   WORD PTR [EBP+10H],101
      JNE   FINISH
;не было ли двойного щелчка мышью?
```

```
      CMP   WORD PTR [EBP+12H],LBN_DBLCLK
      JNE   FINISH
;был двойной щелчок мышью, теперь определим элемент
;получить индекс выбранного элемента
L4:
      PUSH  0
      PUSH  0
      PUSH  LB_GETCURSEL
      PUSH  101
      PUSH  DWORD PTR [EBP+08H]
      CALL  SendDlgItemMessageA@20
;скопировать элемент списка в буфер
      PUSH  OFFSET BUFER
      PUSH  EAX      ; индекс записи
      PUSH  LB_GETTEXT
      PUSH  101
      PUSH  DWORD PTR [EBP+08H]
      CALL  SendDlgItemMessageA@20
;определить, нет ли элемента в правом списке
      PUSH  OFFSET BUFER
      PUSH  -1       ; искать во всем списке
      PUSH  LB_FINDSTRING
      PUSH  102
      PUSH  DWORD PTR [EBP+08H]
      CALL  SendDlgItemMessageA@20
      CMP   EAX,-1
      JNE   FINISH   ; элемент нашли
;не нашли, можно добавлять
      PUSH  OFFSET BUFER
      PUSH  0
      PUSH  LB_ADDSTRING
      PUSH  102
      PUSH  DWORD PTR [EBP+08H]
      CALL  SendDlgItemMessageA@20
      MOV   EAX,-1
      JMP   FIN
L3:
;здесь проверка, не нажата ли клавиша
      CMP   DWORD PTR [EBP+0CH],WM_VKEYTOITEM
      JNE   FINISH
      CMP   WORD PTR [EBP+10H],VK_INSERT
      JE    L4
      MOV   EAX,-1
      JMP   FIN
FINISH:
      MOV   EAX,0
```

```
FIN:
      POP   EDI
      POP   ESI
      POP   EBX
      POP   EBP
      RET   16
WNDPROC  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.5.3:

```
ml /c /coff diallst.asm
rc diallst.rc
link /subsystem:windows diallst.obj diallst.res
```

А теперь комментарий к программе из листинга 2.5.3.

В первую очередь обратите внимание на функцию SendDlgItemMessage. Для посылки сообщения элементам диалогового окна эта функция более удобна, чем SendMessage, т. к. элемент в ней идентифицируется не дескриптором (который еще надо узнать), а номером, определенным в файле ресурсов.

Взглянув на файл ресурсов, вы увидите, что второму (правому) списку присвоено свойство LBS_SORT. Если такое свойство присвоено списку, то при добавлении в него элемента (сообщение LB_ADDSTRING) этот элемент помещается в список так, что список остается упорядоченным. Свойство LBS_SORT стоит системе Windows довольно большой работы. Посредством сообщения WM_COMPAREITEM она определяет нужное положение нового элемента в списке, а затем вставляет его при помощи сообщения LB_INSERTSTRING.

Хотелось бы также обратить внимание на цикл заполнения левого списка. Нам приходится хранить регистр ECX в стеке. Вы скажете: "Дело обыкновенное при организации цикла при помощи команды LOOP". А я вам скажу, что это совсем не очевидно. К сожалению, в документации по функциям API и сообщениям Windows не указывается, какие регистры микропроцессора сохраняются, а какие нет в конкретной функции API. Все это придется устанавливать экспериментально. Известно только, что не должны изменяться регистры EBX, EBP, EDI, ESI.

Сообщение WM_VKEYTOITEM приходит при нажатии какой-либо клавиши, при наличии фокуса на списке. При этом список должен иметь свойство LBS_WANTKEYBOARDINPUT. Именно потому, что данное свойство установлено только у левого списка, у нас нет необходимости проверять, от какого списка пришло сообщение.

Программирование в стиле Windows XP и Windows Vista

Начиная с операционной системы Windows XP, как легко заметить, изменился дизайн окон и управляющих элементов в них. Правда, вы можете при желании вернуться к классическому стилю, так характерному для операционных систем линейки NT (рис. 2.5.1), определив это в окне свойств экрана на вкладке **Оформление**. Не забудьте только, что для того чтобы можно было выбирать стили окон, должна быть запущена служба **Темы** (Themes).

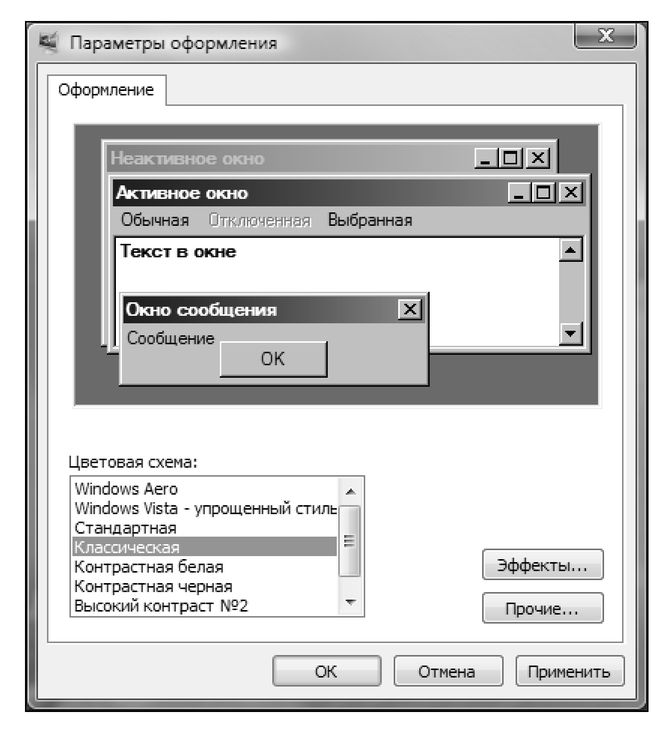


Рис. 2.5.1. Изменение стиля окон и элементов управления в операционной системе Windows Vista

Следует различать оформление окна и стиль изображения элементов управления (рис. 2.5.2).

Если стиль окна автоматически формируется операционной системой, то стиль элементов управления должен определяться при программировании. По этой причине все элементы окон программ, написанных до Windows XP,

имеют априори старый стиль. Однако и при программировании в Windows XP/Vista, если только вы не используете Delphi или C Builder, новый стиль элементов управления тоже сам не появится. Для этого нужно использовать определенную технологию. Вот об этой технологии мы сейчас и поговорим.

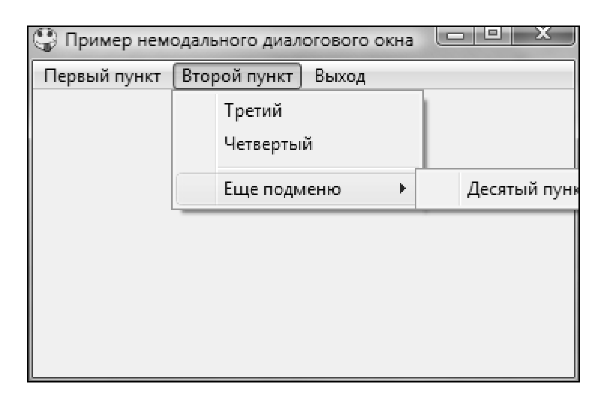


Рис. 2.5.2. Стиль окна и кнопки

Дело в том, что за изображение элементов управления в окне всегда отвечала библиотека user32.dll. Как вы знаете, мы всегда включаем в наши программы строку `includelib ...user32.lib`, чтобы иметь возможность управлять этими элементами в окне. Эта библиотека по-прежнему работает в Windows XP/Vista и изображения элементов управления дает в старом стиле. За новый же стиль элементов в окне теперь отвечает библиотека comctl32.dll. Другими словами, в наши программы следует теперь добавлять еще и строку `inludelib comctl32.lib`. Но этого, однако, недостаточно. Нужно сделать еще следующее:

1. Инициализировать библиотеку comctl32.dll с помощью функции `InitCommonControls` или `InitCommonControlsEx`.
2. Создать специальную структуру — *манифест*, который и сообщит операционной системе, что мы хотим использовать новый стиль изображения элементов управления окна. Этот манифест должен быть написан на языке XML. Вот этот текст (см. листинг 2.5.4).

Листинг 2.5.4. Пример манифеста

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<description>Windows XP program</description>
<assemblyIdentity
  version="1.0.0.0"
```

```
  processorArchitecture="X86"
  name="m.exe"
  type="win32"
/>
<dependency>
  <dependentAssembly>
    <assemblyIdentity
      type="win32"
      name="Microsoft.Windows.Common-Controls"
      version="6.0.0.0"
      processorArchitecture="X86"
      publicKeyToken="6595b64144ccf1df"
      language="*"
    />
  </dependentAssembly>
</dependency>
</assembly>
```

Обратите внимание на строку `version="6.0.0.0"`, которая сообщает операционной системе, что следует использовать функции библиотеки comctl32.dll версии 6.0. Строка `name="m.exe"` задает имя нашей (будущей) программы.

Я думаю, что у читателя сразу возник вопрос: как данный текст можно интегрировать в наш проект? Для этого существуют два способа.

- Если текст нашей программы имеет, скажем, имя m.asm, а сама программа, соответственно, по умолчанию будет называться m.exe, тогда текст на языке XML должен быть помещен в файл с именем m.exe.manifest. Этот файл должен находиться в том же каталоге, что и сам исполняемый модуль m.exe. Этого достаточно, чтобы элементы в окне изображались в новом стиле. Но это не всегда удобно, т. к. файл с расширением manifest может быть случайно испорчен или стерт, поэтому применяют другой более надежный метод.
- XML-текст можно включить в файл ресурсов, поставив в начале файла следующую строку

  ```
  1   24     "m.xml"
  ```

  Предполагается, что XML-текст находится в файле m.xml. Компилятор ресурсов RC.EXE скомпонует этот текст в объектный модуль, откуда он потом будет скопирован в исполняемый модуль.

Теория на этом заканчивается. И мы уже можем писать программы с элементами управления в стиле Windows XP (листинг 2.5.5, рис. 2.5.3).

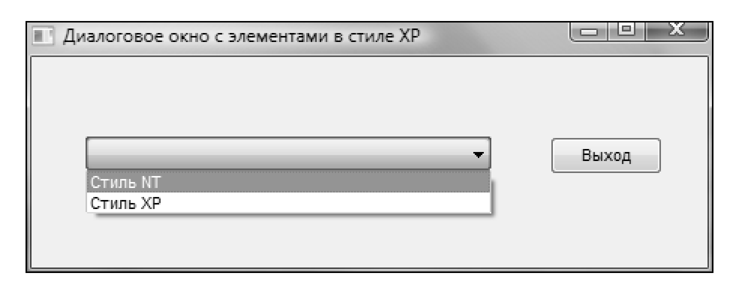


Рис. 2.5.3. Окно, формируемое программой из листинга 2.5.5

Листинг 2.5.5. Простая программа, создающая окно с элементами в стиле Windows XP

```
//файл m.rc
//определение констант
//стиль окна XP
1   24     "m.xml"

#define WS_VSCROLL        200000h
#define CBS_DROPDOWNLIST  3h
#define WS_OVERLAPPED     0h
#define WS_CAPTION        0C00000h
#define IDC_COMBOBOX1     101
#define WS_SYSMENU        0x00080000L
#define WS_MINIMIZEBOX    0x00020000L
#define WS_MAXIMIZEBOX    0x00010000L
//стиль всех элементов в окне
#define WS_CHILD          0x40000000L
//элементы в окне должны быть изначально видимы
#define WS_VISIBLE        0x10000000L
//при помощи клавиши <Tab> можно по очереди активизировать стиль "кнопка"
#define BS_PUSHBUTTON     0x00000000L
//центрировать текст на кнопке
#define BS_CENTER         0x00000300L
#define DS_LOCALEDIT      0x20L

//определение диалогового окна
DIAL1 DIALOG 0, 0, 300, 80
STYLE WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
CAPTION "Диалоговое окно с элементами в стиле XP"
FONT 8, "Arial"
{
 CONTROL "Выход", 5, "button", BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE,
230, 30, 50, 14
 CONTROL "ComboBox1", IDC_COMBOBOX1, "combobox", CBS_DROPDOWNLIST | WS_CHILD |
WS_VISIBLE | WS_VSCROLL, 24, 30, 180, 30
```

```
}
.586P
;плоская модель
.MODEL FLAT, stdcall
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\comctl32.lib
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;константы
;сообщение приходит при закрытии окна
WM_CLOSE        equ 10h
WM_INITDIALOG   equ 110h
WM_COMMAND      equ 111h
;-----------------------------------------------
CB_ADDSTRING    equ 143h
;прототипы внешних процедур
EXTERN  InitCommonControls@0:NEAR
EXTERN  UnregisterHotKey@8:NEAR
EXTERN  RegisterHotKey@16:NEAR
EXTERN  MessageBoxA@16:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  SendMessageA@16:NEAR
EXTERN  GetDlgItem@8:NEAR
EXTERN  MessageBoxA@16:NEAR
EXTERN  SendDlgItemMessageA@20:NEAR
;сегмент данных
_DATA SEGMENT
      HINST  DD 0    ; дескриптор приложения
      PA     DB "DIAL1",0
      STR1   DB "Ошибка!",0
      STR2   DB "Окно сообщения.",0
      S1     DB "Стиль XP",0
      S2     DB "Стиль NT",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
      CALL  InitCommonControls@0
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
```

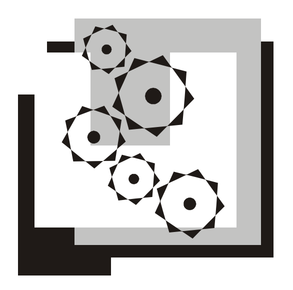
```
;--------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  HINST
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
      PUSH  0
      PUSH  OFFSET STR2
      PUSH  OFFSET STR1
      PUSH  0
      CALL  MessageBoxA@16
KOL:
;--------------------------------
      PUSH  0
      CALL  ExitProcess@4
;--------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;--------------------------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      MOV  EAX,1
      JMP  FIN
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  L2
      PUSH OFFSET S1
      PUSH 0
      PUSH CB_ADDSTRING
      PUSH 101
```

```
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
      PUSH OFFSET S2
      PUSH 0
      PUSH CB_ADDSTRING
      PUSH 101
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
      MOV  EAX,1
      JMP  FIN
L2:
      CMP  DWORD PTR [EBP+0CH],WM_COMMAND
      JNE  FINISH
;кнопка выхода?
      CMP  WORD PTR [EBP+10H],5
      JNE  FINISH
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      MOV  EAX,1
      JMP  FIN
FINISH:
      MOV  EAX,0
FIN:
      POP  EDI
      POP  ESI
      POP  EBX
      POP  EBP
      RET  16
WNDPROC ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.5.5:

```
ML /c /coff m.asm
RC m.rc
LINK /SUBSYSTEM:WINDOWS m.obj m.res
```

Поскольку мы хотели лишь продемонстрировать возможности использования новых стилей изображения элементов управления, то наша программа чрезвычайно проста. В окне имеются лишь два элемента: кнопка и комбинированный список (combobox). Лишний раз подчеркну, что стиль самого окна к нашей теории не имеет никакого отношения, а формируется автоматически операционной системой. Обратим также внимание на то, как добавляются новые элементы в комбинированный список. Для этого используется сообщение `CB_ADDSTRING`.



Глава 2.6

Управление файлами: начало

Управление файлами — важнейшая функциональная составляющая практически любой программы. Начинающие программисты очень много времени уделяют дизайну — окнам, панелям инструментов, кнопкам необычного вида. Но управление файлами важнее — здесь ключ к мастерству.

С появлением скоростных дисков больших объемов значение файлов существенно возросло. Использование API-функций управления файлами может сделать вашу программу более эффективной и производительной. Большинство программ данной главы являются консольными, потому что консольная программа как никакая другая подходит для демонстрации файловой обработки.

Файловая система — это способ скрыть от пользователя и программиста то, что реально происходит при чтении и записи информации во внешней памяти, это логическая надстройка над физической структурой внешней памяти. Операционная система Windows поддерживает две файловые системы для жестких дисков: FAT32 и NTFS (NTFS, New Technology File System). FAT32 является прямой наследницей файловых систем FAT12 и FAT16. Она унаследовала и все их несовершенства, и довольно высокую скорость обработки. Фирма Microsoft оставляет поддержку этой системы в своих новых версиях Windows для совместимости с предыдущими версиями, хотя неоднократно появлялись сообщения, что поддержка FAT32 сведется только к возможности чтения из разделов с этими системами. Файловая система NTFS, напротив, является одной из самых совершенных. Далее мы рассмотрим основные понятия и структуру обеих файловых систем, необходимые для программирования.

Характеристики файлов

Давая описание характеристикам файлов, я буду основываться на тех, которыми манипулируют функции API. О типах и структуре файловых систем

речь пойдет далее. В следующих разделах перечислены основные характеристики файла. Все описанные характеристики поддерживаются файловой системой NTFS, большая часть присутствует и в FAT32.

Атрибут файла

Атрибут файла является целым числом типа `DWORD` и определяет отношение операционной системы к этому файлу.

- `FILE_ATTRIBUTE_READONLY equ 1h`. Атрибут — "только чтение". Приложения могут лишь читать данный файл. Соответственно попытка открыть файл для записи вызовет ошибку. Защита файлов таким атрибутом не является надежной, т. к. можно легко изменить атрибут файла. Это скорее защита для забывчивых программистов.
- `FILE_ATTRIBUTE_HIDDEN equ 2h`. Атрибут — "скрытый файл". "Невидим" при обычном просмотре каталога *(см. разд. "Поиск файлов" далее в этой главе)*.
- `FILE_ATTRIBUTE_SYSTEM equ 4h`. Атрибут — "системный файл". Говорит о том, что данный файл принадлежит операционной системе либо эксклюзивно ею используется.
- `FILE_ATTRIBUTE_DIRECTORY equ 10h`. Атрибут — "каталог". С файлами с таким атрибутом операционная система обращается особым образом, считая его каталогом, т. е. рассматривает его как список файлов, состоящий из записей по 32 байта (длина файла кратна 32). Обычный файл стандартными методами не может быть преобразован в каталог, как и каталог не может быть преобразован к файлу. Для создания каталога используется функция `CreateDirectory`.
- `FILE_ATTRIBUTE_ARCHIVE equ 20h`. Со времен операционной системы MS-DOS таким атрибутом отмечались файлы, над которыми не произведена операция `BACKUP` или `XCOPY`. Для целей программирования данный атрибут эквивалентен нулевому значению атрибута.
- `FILE_ATTRIBUTE_DEVICE equ 40h`. Пока данный атрибут зарезервирован.
- `FILE_ATTRIBUTE_NORMAL equ 80h`. Данный атрибут означает, что у файла не установлены другие атрибуты.
- `FILE_ATTRIBUTE_TEMPORARY equ 100h`. Атрибут означает, что данный файл предназначен для временного хранения данных. После закрытия файла система должна его удалить. Система хранит большую часть такого файла в памяти.

- `FILE_ATTRIBUTE_SPARSE_FILE equ 200h`. Данный атрибут позволяет использовать так называемые распределенные (или разреженные) файлы. Логическая длина таких файлов может много превышать реально занимаемое дисковое пространство. Атрибут появился в файловой системе NTFS 5.0.
- `FILE_ATTRIBUTE_REPARSE_POINT equ 400h`. Атрибут появился в Windows 2000, и предполагается использовать для расширения функциональности файловой системы. Reparse points — это так называемые точки повторной обработки *(см. ниже)*, позволяющие осуществлять на базе файловой системы NTFS технологию иерархического хранения данных (Hierarchical Storage Management). Данная технология позволяет значительно расширить рамки дискового пространства за счет удаленных хранилищ, работа с которыми производится автоматически. Атрибут появился в файловой системе NTFS 5.0.
- `FILE_ATTRIBUTE_COMPRESSED equ 800h`. Для файла это означает, что он сжат системой; для каталога — что вновь создаваемый в нем файл по умолчанию должен быть сжат.
- `FILE_ATTRIBUTE_OFFLINE equ 1000h`. Атрибут означает, что данные файла не доступны в настоящий момент, возможно, находятся на устройстве, в этот момент отключенном.
- `FILE_ATTRIBUTE_NOT_CONTENT_INDEXED equ 2000h`. Файл не может быть проиндексирован службой индексации Windows.
- `FILE_ATTRIBUTE_ENCRYPTED equ 4000h`. Зашифрованный файл. Такой файл предусмотрен в файловой системе NTFS 5.0 *(см. далее)*.

Смену атрибута можно осуществить функцией `SetFileAttributes`, получить значение атрибута функцией `GetFileAttributes`. Значение атрибутов `FILE_ATTRIBUTE_COMPRESSED`, `FILE_ATTRIBUTE_DEVICE`, `FILE_ATTRIBUTE_DIRECTORY`, `FILE_ATTRIBUTE_ENCRYPTED`, `FILE_ATTRIBUTE_REPARSE_POINT`, `FILE_ATTRIBUTE_SPARSE_FILE` можно установить только при помощи функции `DeviceIoControl`.

Следует заметить, что если операционная система не накладывает никаких ограничений на возможности изменения атрибутов файлов, то, в значительной степени, обесценивается смысл самих атрибутов — всегда можно снять атрибут "только для чтения" и делать с файлом все, что заблагорассудится.

Временные характеристики

Здесь и далее под временем мы будем понимать дату и время. Файл имеет три временные характеристики: время создания, время последней модифика-

ции, время последнего доступа. Время отсчитывается в наносекундных интервалах, начиная с 12:00 пополудни 1 января 1600 года, и хранится в двух 32-битных величинах, которые могут быть представлены следующей структурой:

```
FILETIME STRUC
      dwLowDateTime DW ?
      dwLowHighTime DW ?
FILETIME ENDS
```

Надо сказать, что время хранится в так называемых универсальных координатах и должно еще быть преобразовано в локальное время (функция `FileTimeToLocalFileTime`). Получить значение всех трех времен можно функцией `GetFileTime`. Для вывода и манипулирования удобнее иметь дело не с двумя 32-битными величинами, а с более подходящей структурой. Таковой является структура `SYSTIME` (системное время). Она имеет следующий вид:

```
SYSTIME STRUC
      wYear          DW ? ;значение года
      wMonth         DW ? ;номер месяца
      wDayOfWeek     DW ? ;номер дня недели
      wDay           DW ? ;номер дня в месяце
      wHour          DW ? ;час
      wMinute        DW ? ;количество минут
      wSecund        DW ? ;количество секунд
      wMilliseconds  DW ? ;количество миллисекунд
SYSTIME ENDS
```

Для преобразования структуры, получаемой с помощью функции `GetFileTime`, к структуре `SYSTIME` используется API-функция `FileTimeToSystemTime`.

Установить временные характеристики файла можно с помощью функции `SetFileTime`. Для определения времени удобно воспользоваться структурой `SYSTIME`, а потом преобразовать ее к структуре для запуска `SetFileTime` с помощью функции `SystemTimeToFileTime`. Далее будет приведен пример получения временных характеристик файла (см. листинг 2.6.6).

Длина файла

Длина файла в байтах хранится обычно в двух 32-битных величинах либо в одной 64-битной величине. Если 32-битные величины обозначить как `l1` (младшая часть) и `l2` (старшая часть), то 64-битная величина выразится формулой `l2×0FFFFH+l1`. Размер файла можно получить функцией `GetFileSize`. Функция возвращает младшую часть длины файла, в большинстве случаев

этого вполне достаточно. Второй аргумент функции является указателем на старшую часть длины файла. Более удобной функцией получения длины файла является функция `GetFileSizeEx`. Вторым аргументом этой функции является адрес структуры:

```
FSIZE STRUC
      LOWPART  DW ?  ;младшая часть длины
      HIGHPART DW ?  ;старшая часть длины
FSISE ENDS
```

куда и помещается длина файла

Следует заметить, что функции, с помощью которых могут быть получены характеристики файлов, первым своим аргументом имеют дескриптор файла. Другими словами, для того чтобы получить эти характеристики, файл предварительно должен быть открыт (см. далее описание функции `CreateFile`). Альтернативный способ получения этих же параметров — это использовать функцию `FindFirsFile` *(см. разд. "Поиск файлов" далее в этой главе)*.

Имя файла

Кроме указанных характеристик, файл, разумеется, имеет имя. При этом мы будем различать длинное и короткое имена. Точно так же будем различать полный путь (со всеми длинными именами) и укороченный путь (все длинные имена заменены укороченными). Необходимость использования укороченного имени и пути диктуется, прежде всего, тем, что некоторые программы получают путь или имя на стандартный вход и трактуют пробелы как разделители для параметров. Преобразование длинного имени в короткое имя можно осуществить функцией `GetShortPathName`, которая работает и для имени, и для пути. Обратное преобразование можно осуществить функцией `GetFullPathName`.

В данной книге мы не рассматриваем вопроса о прямом доступе к диску. Но вопрос о структуре записей каталога может у читателя все же возникнуть. Это и понятно, ведь с переходом от FAT (MS-DOS) к FAT32[1] (Windows 95), во-первых, появилась возможность хранения файлов с длинным именем, во-вторых, у файла, кроме времени и даты модификации, появились еще время и дата создания и доступа. Где же все это хранится — мы узнаем в следующем разделе.

---

[1] В начале Windows 95 работала с 16-битной FAT, но длинные имена уже поддерживала.

Файловая система FAT32

Для того чтобы ответить на поставленный вопрос, вспомним, что каталог в файловых системах FAT[2] делится на записи длиной 32 байта (см. табл. 2.6.1 и 2.6.2). Пустыми записями считаются записи, содержащие нулевые байты либо начинающиеся с кода `E5H` (для удаленных записей). На файл с обычным именем (8 байтов на имя и 3 — на расширение) отводится 32 байта. В байте со смещением +11 (см. табл. 2.6.1) содержится атрибут файла. Если атрибут файла равен `0FH`, то система считает, что здесь содержится длинное имя. Длинное имя кодируется в Unicode и записывается в обратном порядке перед коротким именем, т. е. за одной или несколькими записями с длинным именем должна следовать запись с обычным именем, содержащим знак ~ (тильда) в седьмой позиции. За знаком тильды в восьмой позиции стоит цифра от 1 и выше. Цифра нужна для того, чтобы различать короткие имена файлов, у которых первые шесть символов имени совпадают. Здесь, в записи с коротким именем, содержится также остальная информация о файле. Как видите, алгоритм просмотра каталога с выявлением информации о файле весьма прост. Обратимся теперь к структуре записи каталога, содержащей короткое имя. В старой операционной системе MS-DOS байты с 12 по 21 никак не использовались системой (см. [1]). Новой системе они пригодились. В табл. 2.6.1 дана новая структура записи каталога.

Таблица 2.6.1. *Новая структура записи каталога*

Смещение	Размер	Содержимое
(+0)	8	Имя файла или каталога, выровненное по левой границе и дополненное пробелами
(+8)	3	Расширение имени файла, выровненное по левой границе и дополненное пробелами
(+11)	1	Атрибут файла
(+12)	2	Используется операционными системами семейства NT. Обеспечивает, в частности, отображение имени файла в правильном регистре
(+14)	2	Время создания файла
(+16)	2	Дата создания файла
(+18)	2	Дата доступа к файлу

[2] FAT (File Allocation Table) — один из элементов, на котором базируются файловые системы MS-DOS и Windows 9*x*. По этой причине часто такие файловые системы называют FAT-системами.

Таблица 2.6.1 *(окончание)*

Смещение	Размер	Содержимое
(+20)	2	Два старших байта в номере первого кластера файла
(+22)	2	Время модификации файла
(+24)	2	Дата модификации файла
(+26)	2	Два младших байта в номере первого кластера файла
(+28)	4	Размер файла в байтах

Как видите, все байты 32-байтной записи каталога теперь заняты. Лишний раз убеждаешься в первоначальной непродуманности файловой системы MS-DOS. Это касается, в частности, длины файла. Как можно заметить, на длину файла отводится всего 4 байта. А как найти длину файла, если на нее требуется более 4 байтов? Разумеется, в этом случае следует считать, что в каталоге хранятся младшие байты длины, а полную длину легко определить, обратившись к таблице размещения файлов. Но, согласитесь, что это уже явная недоработка. Странно также выглядит функция `GetFileSize`, которая возвращает четыре младших байта длины файла, старшие же байты возвращаются во втором параметре функции.

Как я уже сказал, перед записью с коротким именем и характеристиками файла может быть одна или несколько записей с длинным именем. В табл. 2.6.2 представлена структура такой записи.

Таблица 2.6.2. *Структура элемента каталога с длинным именем*

Смещение	Размер	Содержимое
(+0)	1	Порядковый номер фрагмента длинного имени. Используется только 6 битов. В последнем фрагменте к порядковому номеру добавляется число 64
(+1)	10	Пять символов длинного имени в кодировке Unicode (10 байтов)
(+11)	1	Поле атрибута. Всегда должно быть равно `0FH`. Старые DOS-программы игнорируют такие элементы каталога и поэтому видят только короткие имена файлов[3]
(+12)	1	Байт должен быть равен нулю

[3] Неужели разработчики MS-DOS и FAT уже тогда предвидели возможность такого использования атрибута?

***Таблица 2.6.2** (окончание)*

Смещение	Размер	Содержимое
(+13)	1	Контрольная сумма. Предполагалось использовать ее во избежание коллизий при одновременной работе старых DOS-программ и программ для Windows. Проблема может возникнуть, если DOS-программа удалит файл, у которого было длинное имя, и создаст файл, имя которого запишется в каталоге на место удаленного. В этом случае Windows должна будет определить, что длинное имя не относиться к короткому имени файла. Актуальность этого поля давно устарела
(+14)	12	Шесть символов длинного имени в кодировке Unicode
(+26)	2	Два нулевых байта
(+28)	4	Два символа длинного имени в кодировке Unicode

Из табл. 2.6.2 следует, что максимальный размер длинного имени может составить 819 символов ($63 \times 13 = 819$), однако система не позволит установить длину имени, большую чем 260 символов.

Кроме каталогов в рассматриваемой файловой системе имеется еще и собственно FAT — таблица, с помощью которой операционная система отслеживает расположение файла в файловой области. В каталоге хранится начальный кластер файла и длина файла в байтах. Зная размер кластера (в загрузочном[4] секторе раздела хранится информация о размере сектора и количестве секторов в кластере), можно легко определить, сколько кластеров требуется для хранения файла. Этого, однако, не достаточно, т. к. файл может располагаться в нескольких разорванных кластерных цепочках. FAT располагается сразу за резервным сектором раздела, количество которых указано в boot-секторе (Reserved sectors at beginning). Размер элемента FAT[5] составляет 4 байта. Первые два элемента FAT зарезервированы. Первый байт равен всегда `F8H`. Другие три байта могут использоваться по усмотрению операционной системы и недокументированы.

Первый значащий элемент таблицы, таким образом, начинается со второго элемента и соответствует кластеру в файловой области — номера элементов и кластеров совпадают. Вот возможные значения этих элементов:

- 0 — кластер не занят;
- 1 — кластер поврежден;

---

[4] Загрузочный сектор называют еще boot-сектором.

[5] Речь, разумеется, идет о FAT32, в FAT16 размер элемента составляет 2 байта, а в FAT12, которая сейчас используется в файловых системах для дискет, длина элемента составляет 12 бит.

- xFFFFFF8h—xFFFFFFFh — последний кластер в цепочке кластеров файла, старшие 4 бита (x) не учитываются;
- другие значения определяют номер следующего кластера в цепочке для данного файла. При этом 4 старших бита не участвуют в нумерации кластеров. Максимальный номер кластера, таким образом, равен 268 435 447 (FFFFFF7h).

Данная структура таблицы позволяет операционной системе легко по цепочке элементов найти все кластеры данного файла.

Файловая система NTFS

Файловая система NTFS является сложной файловой системой, разработанной независимо от системы FAT. Она является, на мой взгляд, одной из самых совершенных (особенно с точки зрения безопасности) файловых систем, по этой причине я постараюсь изложить ее структуру достаточно подробно[6].

Как и файловой системе FAT, файлы в NTFS записываются в виде последовательности кластеров. Кластеры в файловой системе NTFS могут принимать размеры от 512 байт до 64 Кбайт. Стандартным значением длины кластера является размер 4 Кбайт. Главной структурой NTFS является файл MFT (Master File Table, главная файловая таблица). Надо сказать, что в NTFS нет вообще ничего, кроме файлов — это общая концепция. Имя файла ограничивается 255 символами, а максимальная длина пути не может превышать 32 767 символов.

Рассмотрим рис. 2.6.1, где изображена общая схема размещения файловой системы NTFS на томе.

Для главного файла операционной системы NTFS MFT отводится область в начале раздела (в загрузочной записи раздела при установке помещается номер первого кластера файла MFT), но поскольку это тоже файл, в принципе, он может располагаться где угодно. Для того чтобы не фрагментировать этот файл (хотя он может быть и фрагментирован), для него заранее резервируется объем дискового пространства, составляющий 12% от общего объема раздела. При необходимости операционная система может увеличить или уменьшить эту область, а потом вернуть ее в исходное состояние. По своему смыслу она не является системной областью, поэтому включается в общий объем свободного пространства.

---

[6] Надо сказать, что полное описание файловой системы NTFS Microsoft не раскрывает, поэтому я пользовался лишь отрывочными сведениями, полученными как из источников Microsoft, так и сторонних исследователей.

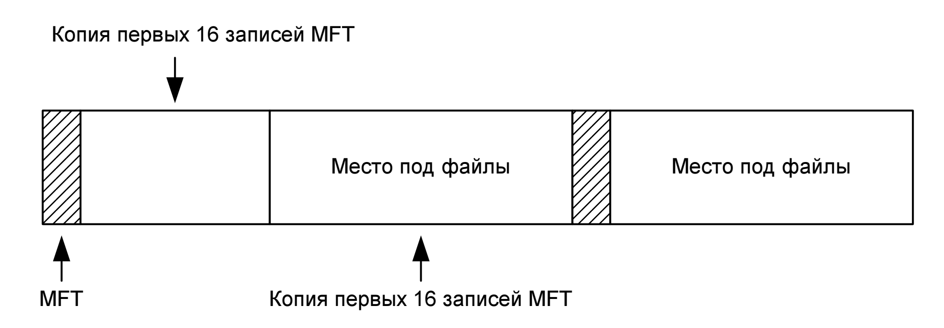


Рис. 2.6.1. Общая структура тома с файловой структурой NTFS

Файл MFT содержит записи о каждом файле системы. Размер записи файла составляет 1 Кбайт. Если при описании файла одной записи не хватает, то используются другие записи. Первые 16 файлов, записи о которых располагаются в начале MFT, являются системными. Принято, что имена этих файлов начинаются с символа `$`. В табл. 2.6.3 представлена информация об этих файлах. Обратите внимание, что в первой (с номером 0) записи файла MFT располагается информация о нем самом[7].

Таблица 2.6.3. *Список метафайлов файловой системы NTFS*

Номер записи в MFT	**Имя системного файла**	**Комментарий**
0	$MFT	Главная файловая таблица
1	$MFTMIRR	Копия первых 16-ти записей главной файловой таблицы. В обычной ситуации этот файл помещается ровно в середине раздела
2	$LOGFILE	Журнал для восстановления системы. Здесь учитываются предстоящие операции. По этому журналу операционная система с большой вероятностью сможет восстановить файловую структуру раздела
3	$VOLUME	Файл тома (метка тома, версия файловой системы, размер и т. п.)
4	$ATTRDEF	Файл содержит список стандартных атрибутов тома
5	$.	Корневой каталог. Как и обычный файл, он может увеличиваться или уменьшаться в размерах. Замечу, что все системные файлы располагаются именно в этом каталоге

[7] Согласитесь, что это красивое решение, создающее определенный избыток информации, позволяющей в определенной ситуации восстановить файловую систему.

Таблица 2.6.3 *(окончание)*

Номер записи в MFT	Имя системного файла	Комментарий
6	$BITMAP	В файле содержится битовый массив учета свободного места на томе
7	$BOOT	Файл начальной загрузки
8	$BADCLUS	Файл, содержащий дефектные блоки
9	$SECURE	Информация о защите
10	$UPCASE	Таблица соответствий заглавных и прописных букв на томе
11	$QUOTA	Каталог, где содержатся файлы, используемые для дисковых квот
12—15		Резервные записи

Обратимся теперь к записи файла MFT. Каждая запись состоит из заголовка, за которым следует заголовок атрибута и его значение. Каждый заголовок содержит: контрольную сумму, порядковый номер файла, увеличивающийся, когда запись используется для другого файла, счетчик обращений к файлу, количество байтов, действительно используемых в записи, и другие поля. За заголовком записи располагается заголовок первого атрибута, а далее значение этого атрибута. Затем идет заголовок второго атрибута и т. д. Если атрибут достаточно велик, то он помещается в отдельном файле (нерезидентный атрибут). Интересно, что если данных в файле немного, то они наоборот хранятся в записи файла MFT.

В табл. 2.6.4 перечислены атрибуты.

Таблица 2.6.4. *Атрибуты записей MFT*

Атрибут	Описание
Стандартная информация (информационный атрибут)	Сведения о владельце, информация о защите, счетчик жестких связей, битовые атрибуты ("только для чтения", "архивный" и т. д.)
Имя файла	Имя файла в кодировке Unicode
Описатель защиты	Этот атрибут устарел. Теперь используется атрибут $EXTENDEDSECURE
Список атрибутов	Расположение дополнительных записей MFT. Используется, если атрибуты не помещаются в записи

Таблица 2.6.4 *(окончание)*

Атрибут	Описание
Идентификатор объекта	64-разрядный идентификатор файла, уникальный для данного тома
Точка повторной обработки	Используется для создания иерархических хранилищ. Наличие этого атрибута предлагает процедуре, анализирующей имя файла, выполнить дополнительные действия
Название тома	Используется в $VOLUME
Информация о томе	Версия тома (используется в $VOLUME)
Корневой индекс	Используется для каталогов
Размещение индекса	Для очень больших каталогов, которые реализуются не в виде обычных списков, а в виде бинарных деревьев (B-деревьев)
Битовый массив	Используется для очень больших каталогов
Поток данных утилиты регистрации	Управляет регистрацией в файле $LOGFILE
Данные	Поток данных файла. Следом за заголовком этого атрибута идет список кластеров, где располагаются данные, либо сами данные, если их объем не превышает несколько сот байтов

Итак, файл в NTFS есть не что иное, как набор атрибутов. *Атрибут* представляется в виде потока байтов (stream)[8]. Как видим, один из атрибутов — это данные, хранящиеся в файле или, как говорят, поток данных. Файловая система допускает добавление файлу новых атрибутов, которые могут содержать какие-то дополнительные данные (*см. главу 2.8*, листинг 2.8.4).

В файловой системе NTFS применено множество интересных технологических решений. Об одной технологии я уже упомянул: небольшой файл целиком помещается в записи файла MFT. Еще один подход предполагает, что операционная система при записи файла старается осуществить операцию таким образом, чтобы было как можно больше цепочек кластеров (участков, где кластеры на диске следуют друг за другом). Группы кластеров файла описываются специальными структурами — записями[9], помещаемыми внутри записи MFT. Так файл, состоящий лишь из одной цепочки кластеров, описы-

[8] См. главу 2.8, листинг 2.8.4 и комментарий к нему.

[9] Не путайте записи файла MFT и файловые записи, описывающие положение кластеров файла на диске.

вается всего одной такой записью. То же можно сказать о файле, состоящем из небольшого числа цепочек. В записи цепочки кластеров описываются парой из двух значений: смещением кластера от начала и количеством кластеров. В заголовке записи указывается смещение первого кластера от начала файла и смещение первого кластера, выходящего за рамки, описываемые данной записью.

На рис. 2.6.2 схематично изображена запись MFT для файла, состоящего всего из 9 кластеров. После заголовка записи, в которой указываются смещения первого кластера в файле и кластера, не охватываемого данной записью, идут две пары чисел, в которых указываются последовательности непрерывно идущих кластеров. Первым элементом пары является смещение кластера от начала дискового пространства и количество кластеров в цепочке. Такие пары называют еще сериями. Как видим, в нашем случае файл состоит из двух непрерывных цепочек и задается двумя сериями. Заметим в этой связи, что числовые значения, определяющие количество кластеров и смещение кластера, являются в операционной системе NTFS 64-битными.

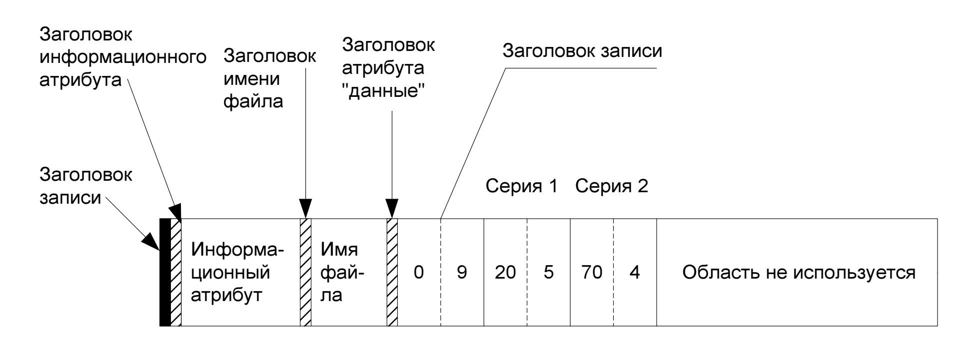


Рис. 2.6.2. Пример записи информации о расположении файла, состоящего всего из девяти кластеров

Что будет, если файл фрагментирован так, что все его цепочки нельзя описать в одной записи файла MFT? В этом случае используются несколько записей MFT. Причем они не обязаны иметь номера, отличающиеся друг от друга на 1. Чтобы связать их друг с другом, используется так называемая базовая запись. В первой записи MFT, описывающей данный файл, она (базовая запись) идет перед записью с описанием кластерных цепочек. Она также имеет заголовок, после которого перечислены номера записей MFT, в которых содержится информация о размещении данных файла на диске. Все остальные записи MFT имеют ту же структуру, которая изображена на рис. 2.6.2. Может возникнуть вопрос: а что если базовая запись не сможет

поместиться в одной записи MFT? В этом случае ее помещают в отдельный файл, т. е. в терминологии NTFS делают нерезидентной.

Рассмотрим еще некоторые особенности файловой системы NTFS.

Каталоги в NTFS

Каталог в файловой системе NTFS представляет собой специфический файл, хранящий ссылки на другие файлы и каталоги, создавая иерархическое строение файловой системы раздела диска. Как и в случае с обычным файлом, если каталог не слишком велик, то он помещается в записи MFT. На рис. 2.6.3 схематически показана запись MFT, содержащая небольшой каталог. Обратите внимание, что в информационном атрибуте содержится информация о корневом каталоге. Сами записи каталога содержат длину имени файла, некоторые другие его параметры, а самое главное содержат номер (индекс) записи MFT для данного файла, в которой хранится уже полная информация о файле. Для больших каталогов используется совсем другой формат хранения. Они строятся в виде бинарных деревьев, что обеспечивает быстрый поиск в алфавитном порядке и ускоряет добавление нового файла.

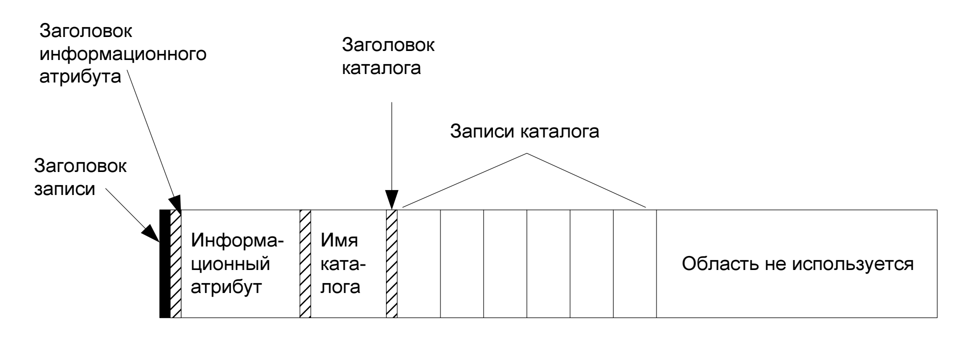


Рис. 2.6.3. Небольшой каталог полностью помещается в записи MFT

Сжатие файлов в NTFS

Файловая система NTFS позволяет хранить файлы в сжатом виде. Механизм также весьма интересен, и о нем следует сказать несколько слов. Сжатие осуществляется блоками, состоящими из 16 кластеров. При записи операционная система пытается сжать вначале первые 16 кластеров, затем следующие и т. д. Если сжать не удалось, то блок записывается как есть. Предположим, что сжимается блок из 16 кластеров. Пусть смещение первого кластера будет 50. Допустим, что сжать удалось на 25%, т. е. из 16 кластеров фактиче-

ски осталось только 12. Для простоты предположим, что кластеры располагаются друг за другом, т. е. представляют цепочку. Сжатая цепочка кластеров в записи MFT будет представлена не одной парой чисел (50,16), а двумя парами (50,12) и (0, 4). Вторая пара необходима, чтобы при чтении файла операционная система могла опознать, какая из цепочек была сжата. Механизм хранения сжатых файлов, как видите, весьма прост и встроен в саму файловую систему.

Стандартным способом получения типа файловой системы в данном разделе является использование функции `GetVolumeInformation`. Мы не будем рассматривать эту функцию, заметим только, что седьмым параметром (из восьми) как раз и является буфер, куда после вызова функции и будет помещен тип файловой системы.

Точки повторной обработки

Точки повторной обработки позволяют наращивать функциональность `NTFS`. Точки повторной обработки появились в версии файловой системы NTFS, предназначенной еще для операционной системы Windows 2000. Было предусмотрено несколько типов таких точек, в том числе: точки монтирования томов, подсоединения каталогов NTFS, управление иерархическими хранилищами данных (HSM, Hierarchical Storage Management).

- *Точки монтирования томов* позволяют привязать том к каталогу, не присваивая символьного обозначения этому тому. Можно, таким образом, под одной буквой объединить несколько томов. Например, если точка монтирования C:\TEMP и к ней подсоединен том D:, то все каталоги этого тома будут доступны через точку монтирования: C:\TEMP\ARH, C:\TEMP\PROGRAM и т. п. При попытке обращения, например к файлу C:\TEMP\PROGRAM\FC.EXE система обнаруживает точку монтирования для каталога TEMP, связанную с томом D:, и далее обращается уже к каталогу PROGRAM этого тома.
- *Точки подсоединения каталогов* очень похожи на точки повторного монтирования. Однако с помощью этого механизма подсоединяются не тома, а каталоги. В предыдущем примере можно подсоединить к каталогу TEMP каталог PROGRAM и аналогичным образом обратиться к файлу FC.EXE: C:\TEMP\PROGRAM\FC.EXE.
- *Управление иерархическим хранилищем данных.* В системе HSM точки повторной обработки служат для миграции редко используемых файлов в резервное хранилище данных. При этом содержимое файла удаляется, а на его место помещается точка повторной обработки. В данных точки повторной обработки содержится информация, используемая системой

HSM для поиска файла на архивном устройстве. При обращении к файлу система по точке повторной обработки определяет, что файл находится в хранилище (и в каком). Запускается механизм перенесения файла из хранилища. После перемещения точка повторной обработки удаляется, а запрос к файлу автоматически повторяется.

Если точка повторной обработки связана с файлом или каталогом, то NTFS создает для нее атрибут с именем `$Reparse`. В этом атрибуте хранятся код и данные точки повторной обработки. Поэтому NTFS без труда обнаруживает все точки повторной обработки на томе.

Чтобы узнать, поддерживает ли ваша система точки повторной обработки, следует вызвать API-функцию `GetVolumeInformation`. Шестой параметр функции представляет собой указатель на переменную типа `DWORD`, которая получает набор флагов файловой системы. Флаг `FILE_SUPPORTS_REPARSE_POINTS = 00000080h` указывает на то, что точки повторной обработки поддерживаются.

Поиск файлов

Для поиска файлов в Windows имеются две функции: `FindFirstFile` и `FindNextFile`. Похожие функции существовали еще в операционной системе MS-DOS. При успешном поиске первая функция возвращает некое число или идентификатор (открытие поиска), который затем используется второй функцией для продолжения поиска.

Первым параметром функции `FindFirstFile` является указатель на строку для поиска файлов, второй параметр — указатель на структуру, которая получает информацию о найденных файлах. Функция `FindNextFile` первым своим параметром имеет идентификатор, полученный первой функцией, а вторым параметром — указатель на структуру, как и в первой функции. Эту структуру можно изучить по программе в листинге 2.6.1.

В листинге 2.6.1 представлена программа, осуществляющая поиск файлов в указанном каталоге. Программа может иметь один или два параметра, или не иметь их вовсе. Если имеются два параметра, то первый параметр трактуется как каталог для поиска, причем программа учитывает, есть ли на конце косая черта или нет (допустимо c:, c:\, c:\windows\, c:\windows\system и т. п.). Второй параметр (в программе он третий, т. к. первым считается командная строка), если он есть, представляет собой маску поиска. Если его нет, то маска поиска берется в виде `*.*`. Наконец, если параметров нет вообще, то поиск осуществляется в текущем каталоге по маске `*.*`. Эту программу легко развить и сделать из нее полезную утилиту. Предоставляю это вам, дорогой читатель. Далее будет дан комментарий к означенной программе.

Листинг 2.6.1. Пример простой программы, которая осуществляет поиск файлов и выводит их название на экран

```
;файл FILES.ASM
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
STD_INPUT_HANDLE equ -10
;прототипы внешних процедур
EXTERN  wsprintfA:NEAR
EXTERN  CharToOemA@8:NEAR
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ReadConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetCommandLineA@0:NEAR
EXTERN  lstrcatA@8:NEAR
EXTERN  FindFirstFileA@8:NEAR
EXTERN  FindNextFileA@8:NEAR
EXTERN  FindClose@4:NEAR
;-----------------------------------------------
;структура, используемая для поиска файла
;при помощи функций FindFirstFile и FindNextFile
_FIND STRUC
;атрибут файла
      ATR      DWORD ?
;время создания файла
      CRTIME   DWORD ?
      DWORD ?
;время доступа к файлу
      ACTIME   DWORD ?
      DWORD ?
;время модификации файла
      WRTIME   DWORD ?
      DWORD ?
;размер файла
      SIZEH    DWORD ? ;старшая часть
      SIZEL    DWORD ? ;младшая часть
;резерв
      DWORD ?
      DWORD ?
;длинное имя файла
      NAM      DB 260 DUP(0)
```

```
;короткое имя файла
      ANAM     DB 14 DUP(0)
_FIND ENDS
;------------------------------------------------
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
      BUF    DB  0
      DB     100 dup(0)
      LENS   DWORD ? ;количество выведенных символов
      HANDL  DWORD ?
      HANDL1 DWORD ?
      MASKA  DB "*.*",0
      AP     DB "\",0
      FIN    _FIND <0>
      TEXT   DB "Для продолжения нажмите клавишу ENTER",13,10,0
      BUFIN  DB 10 DUP(0)
      FINDH  DWORD ?
      NUM    DB 0
      NUMF   DWORD 0 ;счетчик файлов
      NUMD   DWORD 0 ;счетчик каталогов
      FORM   DB "Число найденных файлов: %lu",0
      FORM1  DB "Число найденных каталогов: %lu",0
      BUFER  DB 100 DUP(?)
      DIR    DB " <DIR>",0
      PAR    DB 0 ;количество параметров
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить HANDLE вывода
      PUSH  STD_OUTPUT_HANDLE
      CALL  GetStdHandle@4
      MOV   HANDL,EAX
;получить HANDL1 ввода
      PUSH  STD_INPUT_HANDLE
      CALL  GetStdHandle@4
      MOV   HANDL1,EAX
;преобразовать строки для вывода
      PUSH OFFSET TEXT
      PUSH OFFSET TEXT
      CALL CharToOemA@8
      PUSH OFFSET FORM
```

```
      PUSH OFFSET FORM
      CALL CharToOemA@8
      PUSH OFFSET FORM1
      PUSH OFFSET FORM1
      CALL CharToOemA@8
;получить количество параметров
      CALL NUMPAR
      MOV  PAR,AL
;если параметр один, то искать в текущем каталоге
      CMP  EAX,1
      JE   NO_PAR
;-------------------------------------
;получить параметр с номером EDI
      MOV  EDI,2
      LEA  EBX,BUF
      CALL GETPAR
      PUSH OFFSET BUF
      CALL LENSTR
;если в конце нет "\" - добавим
      CMP  BYTE PTR [BUF+EBX-1],"\"
      JE   NO_PAR
      PUSH OFFSET AP
      PUSH OFFSET BUF
      CALL lstrcatA@8
;нет ли еще параметра, где задана маска поиска
      CMP  PAR,3
      JB   NO_PAR
;получить параметр - маску поиска
      MOV  EDI,3
      LEA  EBX,MASKA
      CALL GETPAR
NO_PAR:
;-------------------------------------
CALL FIND
;вывести количество файлов
      PUSH NUMF
      PUSH OFFSET FORM
      PUSH OFFSET BUFER
      CALL wsprintfA
      LEA  EAX,BUFER
      MOV  EDI,1
      CALL WRITE
;вывести количество каталогов
      PUSH NUMD
      PUSH OFFSET FORM1
      PUSH OFFSET BUFER
```

```
      CALL wsprintfA
      LEA  EAX,BUFER
      MOV  EDI,1
      CALL WRITE
_END:
PUSH 0
      CALL  ExitProcess@4
;***********************
;область процедур
;***********************
;вывести строку (в конце перевод строки)
;EAX - на начало строки
;EDX - с переводом строки или без
WRITE PROC
;получить длину параметра
      PUSH EAX
      CALL LENSTR
      MOV  ESI,EAX
      CMP  EDI,1
      JNE  NO_ENT
;в конце - перевод строки
      MOV  BYTE PTR [EBX+ESI],13
      MOV  BYTE PTR [EBX+ESI+1],10
      MOV  BYTE PTR [EBX+ESI+2],0
      ADD  EBX,2
NO_ENT:
;вывод строки
      PUSH 0
      PUSH OFFSET LENS
      PUSH EBX
      PUSH EAX
      PUSH HANDL
      CALL WriteConsoleA@20
      RET
WRITE ENDP
;процедура определения длины строки
;строка - [EBP+08H]
;длина в EBX
LENSTR PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EAX
      PUSH EDI
;--------------------
      CLD
      MOV EDI,DWORD PTR [EBP+08H]
```

```
      MOV EBX,EDI
      MOV ECX,100 ;ограничить длину строки
      XOR AL,AL
      REPNE SCASB ;найти символ 0
      SUB EDI,EBX ;длина строки, включая 0
      MOV EBX,EDI
      DEC EBX
;--------------------
      POP EDI
      POP EAX
      POP EBP
RET 4
LENSTR ENDP
;процедура определения количества параметров в строке
;определить количество параметров (->EAX)
NUMPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX  ; указатель на строку
      XOR  ECX,ECX  ; счетчик
      MOV  EDX,1    ; признак
L1:
      CMP  BYTE PTR [ESI],0
      JE   L4
      CMP  BYTE PTR [ESI],32
      JE   L3
      ADD  ECX,EDX  ; номер параметра
      MOV  EDX,0
      JMP  L2
L3:
      OR   EDX,1
L2:
      INC  ESI
      JMP  L1
L4:
      MOV  EAX,ECX
RET
NUMPAR ENDP
;получить параметр из командной строки
;EBX - указывает на буфер, куда будет помещен параметр
;в буфер помещается строка с нулем на конце
;EDI - номер параметра
GETPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX  ; указатель на строку
      XOR  ECX,ECX  ; счетчик
      MOV  EDX,1    ; признак
```

```
L1:
      CMP  BYTE PTR [ESI],0
      JE   L4
      CMP  BYTE PTR [ESI],32
      JE   L3
      ADD  ECX,EDX ;номер параметра
      MOV  EDX,0
      JMP  L2
L3:
      OR   EDX,1
L2:
      CMP  ECX,EDI
      JNE  L5
      MOV  AL,BYTE PTR [ESI]
      MOV  BYTE PTR [EBX],AL
      INC  EBX
L5:
      INC  ESI
      JMP  L1
L4:
      MOV  BYTE PTR [EBX],0
      RET
GETPAR ENDP
;поиск в каталоге файлов и их вывод
;имя каталога в BUF
FIND  PROC
;путь с маской
      PUSH OFFSET MASKA
      PUSH OFFSET BUF
      CALL lstrcatA@8
;здесь начало поиска
      PUSH OFFSET FIN
      PUSH OFFSET BUF
      CALL FindFirstFileA@8
      CMP  EAX,-1
      JE   _ERR
;сохранить дескриптор поиска
      MOV  FINDH,EAX
LF:
;исключить "файлы" "." и ".."
      CMP BYTE PTR FIN.NAM,"."
      JE _NO
;не каталог ли?
      TEST BYTE PTR FIN.ATR,10H
      JE   NO_DIR
      PUSH OFFSET DIR
```

```
      PUSH OFFSET FIN.NAM
      CALL lstrcatA@8
      INC  NUMD
      DEC  NUMF
NO_DIR:
;преобразовать строку
      PUSH OFFSET FIN.NAM
      PUSH OFFSET FIN.NAM
      CALL CharToOemA@8
;здесь вывод результата
      LEA  EAX,FIN.NAM
      MOV  EDI,1
      CALL WRITE
;увеличить счетчики
      INC  NUMF
      INC  NUM
;конец страницы?
      CMP  NUM,22
      JNE _NO
      MOV  NUM,0
;ждать ввода строки
      MOV  EDI,0
      LEA  EAX,TEXT
      CALL WRITE
      PUSH 0
      PUSH OFFSET LENS
      PUSH 10
      PUSH OFFSET BUFIN
      PUSH HANDL1
      CALL ReadConsoleA@20
_NO:
;продолжение поиска
      PUSH OFFSET FIN
      PUSH FINDH
      CALL FindNextFileA@8
      CMP  EAX,0
      JNE  LF
;закрыть поиск
      PUSH FINDH
      CALL FindClose@4
_ERR:
      RET
FIND  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.6.1:

```
ml /c /coff   files.asm
link /subsystem:console  files.obj
```

А теперь комментарий к программе из листинга 2.6.1.

Программа довольно проста. Из нового для вас вы обнаружите лишь то, как обращаться с функциями `FindFirstFile` и `FindNextFile`. Процедуры, которые используются для работы с параметрами командной строки, вы уже встречали ранее. Вывод информации осуществляется в текущую консоль, с чем вы также знакомы. Для получения дескриптора консоли используется функция `GetStdHandle`. Процедура `WRITE` позволила несколько упростить те участки программы, которые отвечают за вывод информации на экран. Ранее я обещал, что мы не обойдем вниманием строковые API-функции. В данной программе это обещание выполнено, и наряду со строковыми процедурами "собственного изготовления" используется строковая API-функция `lstrcat`, которая осуществляет сложение (конкатенацию) строк. По поводу параметра в командной строке замечу, что при наличии в имени каталога пробела вам придется задавать имя в укороченном виде. Так, например, вместо `C:\Program Files` придется написать `C:\Progra~1`. Это должно быть понятно — пробелы отделяют параметры. Чтобы корректно решать проблему, необходимо ввести специальный разделитель для параметров, например `-` или `/`, либо использовать кавычки для строки с пробелами. При этом кавычки, разумеется, будут относиться к символам строки-параметра, так что вам придется писать процедуру удаления кавычек из строки.

Данная программа осуществляет поиск в указанном или текущем каталоге. Если бы программа была написана на языке высокого уровня, например C, ее легко можно было бы видоизменить так, чтобы она осуществляла поиск по дереву каталогов. Собственно, небольшая модификация потребовалась бы только для процедуры `FIND`, которая должна была бы вызываться рекурсивно. Можно видеть, что эта легкость произрастает из наличия в языках высокого уровня такого элемента, как локальная переменная. Попробуем осуществить рекурсивный поиск, основываясь на материале *главы 1.2*.

> ***Замечание***
>
> Длина первого параметра API-функции `FindFirstFile` не может превышать значение константы `MAX_PATH`, которая равна 260. Если есть необходимость использовать более длинные строки, то следует обратиться к Unicode-версии данной функции с префиксом `W`. В этом случае длина строки может составлять до 32 000 символов. Только не забудьте осуществить преобразование строки в кодировку Unicode, а также поместить перед именем префикс `\\?\`.

Программа, представленная в листинге 2.6.2, немного похожа на предыдущую программу. Но поиск она осуществляет по дереву каталогов, начиная

с заданного каталога. Эта программа — одна из самых сложных в книге, поэтому советую читателю тщательно в ней разобраться. Может быть, вам удастся ее усовершенствовать. Я могу дать и направление, в котором возможно такое усовершенствование. Дело в том, что вторым параметром командной строки можно указать маску поиска. Если, например, указать маску `*.EXE`, по этой маске будет осуществляться поиск не только файлов, но и каталогов. Этот недостаток и следовало бы устранить в первую очередь.

Поиск по дереву каталогов достаточно просто осуществить рекурсивным образом, однако для этого необходимы локальные переменные[10]. Смысл использования локальной переменной в рекурсивном алгоритме заключается в том, что часть данных должна сохраняться при возврате из процедуры.

В данной программе я, ради простоты, отказался от процедуры `LENSTR` и использую API-функцию `lstrlen`. Кроме того, я усовершенствовал вывод так, чтобы на экран выводилось полное имя файла.

Листинг 2.6.2. Пример программы, которая осуществляет рекурсивный поиск по дереву каталогов

```
;файл FILES.ASM
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE  equ -11
STD_INPUT_HANDLE   equ -10
;прототипы внешних процедур
EXTERN  wsprintfA:NEAR
EXTERN  CharToOemA@8:NEAR
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ReadConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetCommandLineA@0:NEAR
EXTERN  lstrcatA@8:NEAR
EXTERN  lstrcpyA@8:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  FindFirstFileA@8:NEAR
EXTERN  FindNextFileA@8:NEAR
EXTERN  FindClose@4:NEAR
```

[10] Конечно, можно обойтись и без них, храня данные, например, в глобальном массиве, обращаясь к той или иной области массива в зависимости от уровня рекурсии, что по сути будет изобретением велосипеда и при том не самого совершенного.

```
;-----------------------------------------------
;структура, используемая для поиска файла
;при помощи функций FindFirstFile и FindNextFile
_FIND STRUC
;атрибут файла
      ATR     DWORD ?
;время создания файла
      CRTIME  DWORD ?
              DWORD ?
;время доступа к файлу
      ACTIME  DWORD ?
              DWORD ?
;время модификации файла
      WRTIME  DWORD ?
              DWORD ?
;размер файла
      SIZEH   DWORD ?
      SIZEL   DWORD ?
;резерв
              DWORD ?
              DWORD ?
;длинное имя файл
      NAM     DB 260 DUP(0)
;короткое имя файла
      ANAM    DB 14 DUP(0)
_FIND ENDS
;-----------------------------------------------
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      BUF    DB    0
      DB     100 dup(0)
      LENS   DWORD ? ;количество выведенных символов
      HANDL  DWORD ?
      HANDL1 DWORD ?
      MASKA  DB "*.*"
      DB     50 DUP(0)
      AP     DB "\",0
      FIN    _FIND <0>
      TEXT   DB "Для продолжения нажмите клавишу ENTER",13,10,0
      BUFIN  DB 10 DUP(0) ;буфер ввода
      NUM    DB 0
      NUMF   DWORD 0 ;счетчик файлов
```

```
      NUMD   DWORD 0 ;счетчик каталогов
      FORM   DB "Число найденных файлов: %lu",0
      FORM1  DB "Число найденных каталогов: %lu",0
      DIRN   DB " <DIR>",0
      PAR    DWORD 0
      PRIZN  DB 0
_DATA ENDS
;сегмент кода
_TEXT  SEGMENT
START:
;получить HANDLE вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;получить HANDL1 ввода
      PUSH STD_INPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL1,EAX
;преобразовать строки для вывода
      PUSH OFFSET TEXT
      PUSH OFFSET TEXT
      CALL CharToOemA@8
      PUSH OFFSET FORM
      PUSH OFFSET FORM
      CALL CharToOemA@8
      PUSH OFFSET FORM1
      PUSH OFFSET FORM1
      CALL CharToOemA@8
;получить количество параметров
      CALL NUMPAR
      MOV  PAR,EAX
;если параметр один, то искать в текущем каталоге
      CMP   EAX,1
      JE    NO_PAR
;-------------------------------------
;получить параметр с номером EDI
      MOV   EDI,2
      LEA   EBX,BUF
      CALL  GETPAR
      CMP   PAR,3
      JB    NO_PAR
;получить параметр - маску поиска
      MOV   EDI,3
      LEA   EBX,MASKA
      CALL  GETPAR
```

```
NO_PAR:
;------------------------------------
      PUSH  OFFSET BUF
      CALL  FIND
;вывести количество файлов
      PUSH NUMF
      PUSH OFFSET FORM
      PUSH OFFSET BUF
      CALL wsprintfA
      LEA  EAX,BUF
      MOV  EDI,1
      CALL WRITE
;++++++++++++++++
;вывести количество каталогов
      PUSH NUMD
      PUSH OFFSET FORM1
      PUSH OFFSET BUF
      CALL wsprintfA
      LEA  EAX,BUF
      MOV  EDI,1
      CALL WRITE
_END:
      PUSH  0
      CALL  ExitProcess@4
;область процедур
;****************************************
;вывести строку (в конце перевод строки)
;EAX - на начало строки
;EDX - с переводом строки или без
WRITE   PROC
;получить длину параметра
      PUSH  EAX
      PUSH  EAX
      CALL  lstrlenA@4
      MOV   ESI,EAX
      POP   EBX
      CMP   EDI,1
      JNE   NO_ENT
;в конце - перевод строки
      MOV  BYTE PTR [EBX+ESI],13
      MOV  BYTE PTR [EBX+ESI+1],10
      MOV  BYTE PTR [EBX+ESI+2],0
      ADD  EAX,2
NO_ENT:
;вывод строки
      PUSH 0
```

```
        PUSH OFFSET LENS
        PUSH EAX
        PUSH EBX
        PUSH HANDL
        CALL WriteConsoleA@20
        RET
WRITE   ENDP
;процедура определения количества параметров в строке
;определить количество параметров (->EAX)
NUMPAR  PROC
        CALL GetCommandLineA@0
        MOV  ESI,EAX ;указатель на строку
        XOR  ECX,ECX ;счетчик
        MOV  EDX,1   ;признак
L1:
        CMP  BYTE PTR [ESI],0
        JE   L4
        CMP  BYTE PTR [ESI],32
        JE   L3
        ADD  ECX,EDX   ;номер параметра
        MOV  EDX,0
        JMP  L2
L3:
        OR   EDX,1
L2:
        INC  ESI
        JMP  L1
L4:
        MOV  EAX,ECX
        RET
NUMPAR  ENDP
;получить параметр из командной строки
;EBX - указывает на буфер, куда будет помещен параметр
;в буфер помещается строка с нулем на конце
;EDI - номер параметра
GETPAR PROC
        CALL GetCommandLineA@0
        MOV  ESI,EAX ;указатель на строку
        XOR  ECX,ECX ;счетчик
        MOV  EDX,1   ;признак
L1:
        CMP  BYTE PTR [ESI],0
        JE   L4
        CMP  BYTE PTR [ESI],32
        JE   L3
        ADD  ECX,EDX  ;номер параметра
```

```
      MOV  EDX,0
      JMP  L2
L3:
      OR   EDX,1
L2:
      CMP  ECX,EDI
      JNE  L5
      MOV  AL,BYTE PTR [ESI]
      MOV  BYTE PTR [EBX],AL
      INC  EBX
L5:
      INC  ESI
      JMP  L1
L4:
      MOV  BYTE PTR [EBX],0
      RET
GETPAR ENDP
;-----------------------------------
;поиск в каталоге файлов и их вывод
FINDH   EQU  [EBP-4]   ;дескриптор поиска
DIRS    EQU  [EBP-304] ;полное имя файла
DIRSS   EQU  [EBP-604] ;для хранения каталога
DIRV    EQU  [EBP-904] ;для временного хранения
DIR     EQU  [EBP+8]   ;параметр - имя каталога
FIND  PROC
      PUSH EBP
      MOV  EBP,ESP
      SUB  ESP,904
;инициализация локальных переменных
      MOV   ECX,300
      MOV   AL,0
      MOV   EDI,0
CLR:
      MOV   BYTE PTR DIRS+[EDI],AL
      MOV   BYTE PTR DIRSS+[EDI],AL
      MOV   BYTE PTR DIRV+[EDI],AL
      INC   EDI
      LOOP  CLR
;определить длину пути
      PUSH  DIR
      CALL  lstrlenA@4
      MOV   EBX,EAX
      MOV   EDI,DIR
      CMP   BYTE PTR [EDI],0
      JE    _OK
```

```
;если в конце нет "\" - добавим
      CMP   BYTE PTR [EDI+EBX-1],"\"
      JE    _OK
      PUSH  OFFSET AP
      PUSH  DIR
      CALL  lstrcatA@8
_OK:
;запомним каталог
      PUSH  DIR
      LEA   EAX,DIRSS
      PUSH  EAX
      CALL  lstrcpyA@8
;путь с маской
      PUSH  OFFSET MASKA
      PUSH  DIR
      CALL  lstrcatA@8
;здесь начало поиска
      PUSH  OFFSET FIN
      PUSH  DWORD PTR DIR
      CALL  FindFirstFileA@8
      CMP   EAX,-1
      JE   _ERR
;сохранить дескриптор поиска
      MOV  FINDH,EAX
LF:
;исключить "файлы" "." и ".."
      CMP  BYTE PTR FIN.NAM,"."
      JE   _FF
;-------------------
      LEA   EAX,DIRSS
      PUSH  EAX
      LEA   EAX,DIRS
      PUSH  EAX
      CALL  lstrcpyA@8
;-------------------
      PUSH OFFSET FIN.NAM
      LEA  EAX,DIRS
      PUSH EAX
      CALL lstrcatA@8
;не каталог ли?
      TEST BYTE PTR FIN.ATR,10H
      JE   NO_DIR
;добавить в строку <DIR>
      PUSH OFFSET DIRN
      LEA  EAX,DIRS
      PUSH EAX
      CALL lstrcatA@8
```

```
;увеличим счетчики
      INC  NUMD
      DEC  NUMF
;установим признак каталога
      MOV  PRIZN,1
;вывести имя каталога
      LEA  EAX,DIRS
      PUSH EAX
      CALL OUTF
      JMP  _NO
NO_DIR:
;вывести имя файла
      LEA  EAX,DIRS
      PUSH EAX
      CALL OUTF
;признак файла (не каталога)
      MOV  PRIZN,0
_NO:
      CMP  PRIZN,0
      JZ   _F
;каталог, готовимся к рекурсивному вызову
      LEA   EAX,DIRSS
      PUSH  EAX
      LEA   EAX,DIRV
      PUSH  EAX
      CALL  lstrcpyA@8
      PUSH  OFFSET FIN.NAM
      LEA   EAX,DIRV
      PUSH  EAX
      CALL  lstrcatA@8
;осуществляем вызов
      LEA   EAX,DIRV
      PUSH  EAX
      CALL FIND
;продолжение поиска
_F:
      INC  NUMF
_FF:
      PUSH OFFSET FIN
      PUSH FINDH
      CALL FindNextFileA@8
      CMP  EAX,0
      JNE  LF
;закрыть дескриптор поиска
      PUSH FINDH
      CALL FindClose@4
```

```
_ERR:
      MOV  ESP,EBP
      POP  EBP
      RET  4
FIND  ENDP
;---------------------------
;страничный вывод имен найденных файлов
STRN   EQU  [EBP+8]
OUTF  PROC
      PUSH EBP
      MOV  EBP,ESP
;преобразовать строку
      PUSH STRN
      PUSH STRN
      CALL CharToOemA@8
;здесь вывод результата
      MOV  EAX,STRN
      MOV  EDI,1
      CALL WRITE
      INC  NUM
;конец страницы?
      CMP  NUM,22
      JNE  NO
      MOV  NUM,0
;ждать ввод строки
      MOV  EDI,0
      LEA  EAX,TEXT
      CALL WRITE
      PUSH 0
      PUSH OFFSET LENS
      PUSH 10
      PUSH OFFSET BUFIN
      PUSH HANDL1
      CALL ReadConsoleA@20
NO:
      POP EBP
      RET 4
OUTF  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.6.2:

```
ml /c /coff   files.asm
link /subsystem:console  files.obj
```

Прокомментирую программу из листинга 2.6.2.

Выясним, какую роль играют локальные переменные в процедуре `FIND` (см. листинг 2.6.2). В переменной `FINDH` хранится дескриптор поиска в данном каталоге. Рекурсивный вызов процедуры `FIND` может происходить и тогда, когда поиск в текущем каталоге еще не закончился. Следовательно, после возврата из рекурсии поиск должен быть продолжен. Это можно обеспечить только старым значением дескриптора. Локальная переменная обеспечивает такую возможность, поскольку она разрушается (точнее, для нее отводится новая область памяти) только при переходе на более низкий уровень (к родительскому каталогу).

Аналогичную роль играет переменная `DIRSS`. В ней хранится текущий каталог. Это важно, т. к. с помощью этой переменной формируется полное имя файла.

Переменные `DIRS` и `DIRV` играют вспомогательные роли. В принципе, вместо них можно было бы использовать и глобальные переменные. Тем более, что с точки зрения эффективности рекурсивных алгоритмов, чем меньше объем локальных переменных — тем лучше.

Еще один вопрос я хочу здесь обсудить. Для передачи имени каталога при вызове процедуры используется переменная `DIRV`. Почему же для этой цели нельзя использовать переменную `DIRSS`? Причина вот в чем. В процедуру передается не само значение, а указатель (адрес). Следовательно, любые изменения с параметром `DIR` приведут к аналогичным изменениям с переменной `DIRSS` на нижнем уровне рекурсии. В чем мы, разумеется, не заинтересованы.

Приемы работы с двоичными файлами

Манипуляции внешними файлами[11] основаны на нескольких функциях API, главной и наиболее сложной из которых является функция `CreateFile`. Мы отложим подробное описание данной функции до *главы 2.8*, здесь же будет дана практическая информация, необходимая для того, чтобы начать использовать эту функцию в своих программах. Замечу, однако, что с помощью этой функции можно не только создавать или открывать файл, но и манипулировать такими объектами, как каналы (pipes), консоли, устройства жесткого диска (disk device), коммуникационный ресурс и др. *(см. подробно главу 2.8)*. Функция различает устройство по структуре имени. К примеру, `"C:\config.sys"` определяет файл, а `CONOUT$` — буфер вывода текущей консоли и т. д.

[11] Имеется в виду файлами, расположенными на внешнем устройстве.

Сейчас я представлю две простые, но весьма важные программы (листинги 2.6.3 и 2.6.4). Обе программы выводят содержимое текстового файла[12], имя которого указано в командной строке, в текущую консоль. В первом случае мы получаем дескриптор текущей консоли стандартным способом. Во втором случае (листинг 2.6.4) открываем консоль как файл и, соответственно, выводим туда информацию, как в файл. Хочу обратить ваше внимание на роль буфера, в который читается содержимое файла. Поэкспериментируйте с размером буфера, взяв для пробы большой текстовый файл. Интересно, что в указанных программах никак не учитывается структура текстового файла. Для такого ввода/вывода это ни к чему. Далее мы поговорим и о структуре текстовых файлов.

Листинг 2.6.3. Вывод на консоль содержимого текстового файла. Первый способ

```
;файл FILES1.ASM
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
GENERIC_READ      equ 80000000h
GENERIC_WRITE     equ 40000000h
GEN = GENERIC_READ or GENERIC_WRITE
SHARE = 0
OPEN_EXISTING     equ 3
;прототипы внешних процедур
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetCommandLineA@0:NEAR
EXTERN  CreateFileA@28:NEAR
EXTERN  CloseHandle@4:NEAR
EXTERN  ReadFile@20:NEAR
;-----------------------------------------------
;директивы компоновщику для подключения библиотек
;includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      HANDL DWORD ?
```

[12] Точнее, любого файла, но смысл выводить файл на консоль именно таким образом имеется только для текстового файла.

```
      HFILE DWORD ?
      BUF   DB 100 DUP(0)
      BUFER DB 300 DUP(0)
      NUMB  DWORD ?
      NUMW  DWORD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить HANDLE вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;получить количество параметров
      CALL NUMPAR
      CMP  EAX,1
      JE   NO_PAR
;-------------------------------------
;получить параметр с номером EDI
      MOV  EDI,2
      LEA  EBX,BUF
      CALL GETPAR
;открыть файл
      PUSH 0                  ; должен быть равен 0
      PUSH 0                  ; атрибут файла (если создаем)
      PUSH OPEN_EXISTING      ; как открывать
      PUSH 0                  ; указатель на security attr
      PUSH 0                  ; режим общего доступа
      PUSH GEN                ; режим доступа
      PUSH OFFSET BUF         ; имя файла
      CALL CreateFileA@28
      CMP  EAX,-1
      JE   NO_PAR
      MOV  HFILE,EAX
LOO:
;прочесть в буфер
      PUSH 0
      PUSH OFFSET NUMB
      PUSH 300
      PUSH OFFSET BUFER
      PUSH HFILE
      CALL ReadFile@20
;вывести содержимое буфера на консоль
      PUSH 0
      PUSH OFFSET NUMW
      PUSH NUMB
```

```
      PUSH OFFSET BUFER
      PUSH HANDL
      CALL WriteConsoleA@20
;проверить, не последние ли байты прочитаны
      CMP NUMB,300
      JE LOO
;закрыть файл
      PUSH HFILE
      CALL CloseHandle@4
;конец работы программы
NO_PAR:
      PUSH 0
      CALL ExitProcess@4
;область процедур
;процедура определения количества параметров в строке
;определить количество параметров (->EAX)
NUMPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX   ; указатель на строку
      XOR  ECX,ECX   ; счетчик
      MOV  EDX,1     ; признак
L1:
      CMP BYTE PTR [ESI],0
      JE  L4
      CMP BYTE PTR [ESI],32
      JE  L3
      ADD ECX,EDX    ; номер параметра
      MOV EDX,0
      JMP L2
L3:
      OR EDX,1
L2:
      INC ESI
      JMP L1
L4:
      MOV EAX,ECX
      RET
NUMPAR ENDP
;получить параметр из командной строки
;EBX - указывает на буфер, куда будет помещен параметр
;в буфер помещается строка с нулем на конце
;EDI - номер параметра
GETPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX  ; указатель на строку
      XOR  ECX,ECX  ; счетчик
      MOV  EDX,1    ; признак
```

```
L1:
      CMP  BYTE PTR [ESI],0
      JE   L4
      CMP  BYTE PTR [ESI],32
      JE   L3
      ADD  ECX,EDX  ; номер параметра
      MOV  EDX,0
      JMP  L2
L3:
      OR   EDX,1
L2:
      CMP ECX,EDI
      JNE L5
      MOV AL,BYTE PTR [ESI]
      MOV BYTE PTR [EBX],AL
      INC EBX
L5:
      INC ESI
      JMP L1
L4:
      MOV BYTE PTR [EBX],0
      RET
GETPAR ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.6.3:

```
ml /c /coff   files1.asm
link /subsystem:console  files1.obj
```

Обратите внимание на прием, который используется для вывода файла (см. листинг 2.6.3). Организуется "бесконечный цикл", в котором поочередно читается и записывается порция информации из файла. Признаком окончания процесса является тот факт, что в очередной раз буфер для чтения оказался заполненным не целиком. Причем выходной параметр для функции чтения — количество прочитанных байтов — оказывается входным параметром для функции записи — количество байтов, которое следует записать.

Листинг 2.6.4. Вывод на консоль содержимого текстового файла. Второй способ

```
;файл FILES2.ASM
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
```

```
STD_OUTPUT_HANDLE equ -11
GENERIC_READ      equ 80000000h
GENERIC_WRITE     equ 40000000h
GEN = GENERIC_READ or GENERIC_WRITE
SHARE = 0
OPEN_EXISTING     equ 3
;прототипы внешних процедур
EXTERN  ExitProcess@4:NEAR
EXTERN  GetCommandLineA@0:NEAR
EXTERN  CreateFileA@28:NEAR
EXTERN  CloseHandle@4:NEAR
EXTERN  ReadFile@20:NEAR
EXTERN  WriteFile@20:NEAR
;------------------------------------------------
;директивы компоновщику для подключения библиотек
;includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
      HANDL  DWORD ?
      HFILE  DWORD ?
      BUF    DB 100 DUP(0)
      BUFER  DB 300 DUP(0)
      NUMB   DWORD ?
      NUMW   DWORD ?
      NAMEOUT DB "CONOUT$"
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить HANDLE вывода (консоли) как файла
      PUSH 0
      PUSH 0
      PUSH OPEN_EXISTING
      PUSH 0
      PUSH 0
      PUSH GEN
      PUSH OFFSET NAMEOUT
      CALL CreateFileA@28
      MOV  HANDL,EAX
;получить количество параметров
      CALL NUMPAR
      CMP  EAX,1
      JE   NO_PAR
;--------------------------------------
```

```
;получить параметр номером EDI
      MOV  EDI,2
      LEA  EBX,BUF
      CALL GETPAR
;открыть файл
      PUSH 0
      PUSH 0
      PUSH OPEN_EXISTING
      PUSH 0
      PUSH 0
      PUSH GEN
      PUSH OFFSET BUF
      CALL CreateFileA@28
      CMP  EAX,-1
      JE   NO_PAR
      MOV  HFILE,EAX
LOO:
;прочесть в буфер
      PUSH 0
      PUSH OFFSET NUMB
      PUSH 300
      PUSH OFFSET BUFER
      PUSH HFILE
      CALL ReadFile@20
;вывести на консоль как в файл
      PUSH 0
      PUSH OFFSET NUMW
      PUSH NUMB
      PUSH OFFSET BUFER
      PUSH HANDL
      CALL WriteFile@20
      CMP  NUMB,300
      JE   LOO
;закрыть файл
      PUSH HFILE
      CALL CloseHandle@4
;конец работы программы
NO_PAR:
      PUSH 0
      CALL ExitProcess@4
;область процедур
;процедура определения количества параметров в строке
;определить количество параметров (->EAX)
NUMPAR PROC
        CALL GetCommandLineA@0
        MOV  ESI,EAX  ; указатель на строку
```

```
        XOR  ECX,ECX  ; счетчик
        MOV  EDX,1    ; признак
L1:
        CMP BYTE PTR [ESI],0
        JE  L4
        CMP BYTE PTR [ESI],32
        JE  L3
        ADD ECX,EDX ;номер параметра
        MOV EDX,0
        JMP L2
L3:
        OR EDX,1
L2:
        INC ESI
        JMP L1
L4:
        MOV EAX,ECX
        RET
NUMPAR ENDP
;получить параметр из командной строки
;EBX — указывает на буфер, куда будет помещен параметр
;в буфер помещается строка с нулем на конце
;EDI — номер параметра
GETPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX  ; указатель на строку
      XOR  ECX,ECX  ; счетчик
      MOV  EDX,1    ; признак
L1:
      CMP BYTE PTR [ESI],0
      JE  L4
      CMP BYTE PTR [ESI],32
      JE  L3
      ADD ECX,EDX ;номер параметра
      MOV EDX,0
      JMP L2
L3:
      OR EDX,1
L2:
      CMP ECX,EDI
      JNE L5
      MOV AL,BYTE PTR [ESI]
      MOV BYTE PTR [EBX],AL
      INC EBX
L5:
      INC ESI
      JMP L1
```

```
L4:
     MOV BYTE PTR [EBX],0
     RET
GETPAR ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.6.4:

```
ml /c /coff   files2.asm
link /subsystem:console  files2.obj
```

Сейчас мы поговорим более подробно о структуре текстового файла. При работе с языками высокого уровня теряются определенные алгоритмические навыки. Это касается, в частности, и работы с текстовыми файлами. Ассемблер не дает расслабиться. Рассмотрим возможные варианты работы с текстовыми файлами.

Основным признаком текстового файла является то, что он состоит из строк разной длины. Строки отделены друг от друга разделителями. Чаще всего это последовательность двух кодов — 13 и 10. Возможны и другие варианты, например, некоторые старые редакторы отделяли строки только одним кодом 13 (так, например, принято и в операционной системе UNIX).

Построчное чтение текстового файла можно осуществить четырьмя наиболее очевидными способами.

- *Побайтное чтение из файла.* Как только достигаем символа-разделителя, производим действие над считанной строкой и переходим к чтению следующей строки. При этом, разумеется, следует учесть, что на конце файла может не быть символа-разделителя. Если кто-то решит, что это слишком медленный способ, то замечу, что Windows неплохо кэширует диск, поэтому все выглядит не так уж плохо.
- *Чтение в небольшой буфер.* Буфер, однако, должен иметь достаточную длину, так чтобы туда входила, по крайней мере, одна строка. Прочитав, находим в буфере конец строки и производим над ней какое-либо действие. Далее следует обратиться к файлу и передвинуть указатель так, чтобы он был в файле на начале следующей строки и, разумеется, повторить действие. Минусом данного подхода является то, что метод даст ошибку, если длина строки окажется больше длины буфера.
- *Чтение в буфер произвольной длины.* После чтения производится поиск всех строк, попавших в буфер, и выполнение над ними действий. При этом с большой вероятностью должна возникнуть ситуация, когда одна строка не полностью умещается в буфере. Мы обязаны учесть такую воз-

можность. Замечу, что данный метод должен правильно работать и в случае, когда в буфер не помещается и одна строка.

- *Чтение в буфер, в который помещается весь файл.* Это частный случай третьего подхода и наиболее простой с точки зрения программирования.

В программе из листинга 2.6.5 реализуется третий подход.

Листинг 2.6.5. Пример обработки текстового файла

```
;файл FILES3.ASM
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
GENERIC_READ      equ 80000000h
GENERIC_WRITE     equ 40000000h
GEN = GENERIC_READ or GENERIC_WRITE
SHARE = 0
OPEN_EXISTING     equ 3
;прототипы внешних процедур
EXTERN  ExitProcess@4:NEAR
EXTERN  GetCommandLineA@0:NEAR
EXTERN  CreateFileA@28:NEAR
EXTERN  CloseHandle@4:NEAR
EXTERN  ReadFile@20:NEAR
EXTERN  WriteFile@20:NEAR
EXTERN  CharToOemA@8:NEAR
;------------------------------------------------
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
      HANDL   DWORD ? ;дескриптор консоли
      HFILE   DWORD ? ;дескриптор файла
      BUF     DB 100 DUP(0) ;буфер для параметров
      BUFER   DB 1000 DUP(0);буфер для файла
      NAMEOUT DB "CONOUT$"
      INDS    DD 0 ;номер символа в строке
      INDB    DD 0 ;номер символа в буфере
      NUMB    DD ?
      NUMC    DD ?
      PRIZN   DD 0
      STROKA  DB 300 DUP(0)
```

```
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить HANDLE вывода (консоли) как файла
      PUSH 0
      PUSH 0
      PUSH OPEN_EXISTING
      PUSH 0
      PUSH 0
      PUSH GEN
      PUSH OFFSET NAMEOUT
      CALL CreateFileA@28
      MOV  HANDL,EAX
;получить количество параметров
      CALL NUMPAR
      CMP  EAX,1
      JE   NO_PAR
;-------------------------------------
;получить параметр с номером EDI
      MOV  EDI,2
      LEA  EBX,BUF
      CALL GETPAR
;открыть файл
      PUSH 0
      PUSH 0
      PUSH OPEN_EXISTING
      PUSH 0
      PUSH 0
      PUSH GEN
      PUSH OFFSET BUF
      CALL CreateFileA@28
      CMP  EAX,-1
      JE   NO_PAR
      MOV  HFILE,EAX
;++++++++++++++++++++++++++++++
LOO:
;читать 1000 байтов
      PUSH 0
      PUSH OFFSET NUMB
      PUSH 1000
      PUSH OFFSET BUFER
      PUSH HFILE
      CALL ReadFile@20
      MOV  INDB,0
      MOV  INDS,0
```

```
;проверим, есть ли в буфере байты
      CMP  NUMB,0
      JZ  _CLOSE
;заполняем строку
LOO1:
      MOV  EDI,INDS
      MOV  ESI,INDB
      MOV  AL,BYTE PTR BUFER[ESI]
      CMP  AL,13 ;проверка на конец строки
      JE  _ENDSTR
      MOV  BYTE PTR STROKA[EDI],AL
      INC  ESI
      INC  EDI
      MOV  INDS,EDI
      MOV  INDB,ESI
      CMP  NUMB,ESI ;проверка на конец буфера
      JNBE LOO1
;закончился буфер
      MOV  INDS,EDI
      MOV  INDB,ESI
      JMP  LOO
_ENDSTR:
;делаем что-то со строкой
      CALL OUTST
;обнулить строку
      MOV  INDS,0
;перейти к следующей строке в буфере
      ADD  INDB,2
;не закончился ли буфер?
      MOV  ESI,INDB
      CMP  NUMB,ESI
      JAE  LOO1
      JMP  LOO
;+++++++++++++++++++++++++++++
_CLOSE:
;проверим, не пустая ли строка
      CMP  INDS,0
      JZ   CONT
;делаем что-то со строкой
      CALL OUTST
CONT:
;закрыть файлы
      PUSH  HFILE
      CALL  CloseHandle@4
;конец работы программы
```

```
NO_PAR:
      PUSH  0
      CALL  ExitProcess@4
;область процедур
;процедура определения количества параметров в строке
;определить количество параметров (->EAX)
NUMPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX  ; указатель на строку
      XOR  ECX,ECX  ; счетчик
      MOV  EDX,1    ; признак
L1:
      CMP  BYTE PTR [ESI],0
      JE   L4
      CMP  BYTE PTR [ESI],32
JE L3
      ADD  ECX,EDX   ; номер параметра
      MOV  EDX,0
      JMP  L2
L3:
      OR   EDX,1
L2:
      INC  ESI
      JMP  L1
L4:
      MOV  EAX,ECX
      RET
NUMPAR ENDP
;получить параметр из командной строки
;EBX - указывает на буфер, куда будет помещен параметр
;в буфер помещается строка с нулем на конце
;EDI - номер параметра
GETPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX  ; указатель на строку
      XOR  ECX,ECX  ; счетчик
      MOV  EDX,1    ; признак
L1:
      CMP  BYTE PTR [ESI],0
      JE   L4
      CMP  BYTE PTR [ESI],32
      JE   L3
      ADD  ECX,EDX  ; номер параметра
      MOV  EDX,0
      JMP  L2
L3:
      OR   EDX,1
```

```
L2:
      CMP  ECX,EDI
      JNE  L5
      MOV  AL,BYTE PTR [ESI]
      MOV  BYTE PTR [EBX],AL
      INC  EBX
L5:
      INC  ESI
      JMP  L1
L4:
      MOV  BYTE PTR [EBX],0
      RET
GETPAR ENDP
;вывести строку в консоль с разделителем
OUTST PROC
      MOV  EBX,INDS
      MOV  BYTE PTR STROKA[EBX],0
      PUSH OFFSET STROKA
      PUSH OFFSET STROKA
      CALL CharToOemA@8
;в конце строки - разделитель
      MOV  BYTE PTR STROKA[EBX],10
      INC  INDS
;вывести строку
      PUSH 0
      PUSH OFFSET NUMC
      PUSH INDS
      PUSH OFFSET STROKA
      PUSH HANDL
      CALL WriteFile@20
      RET
OUTST ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.6.5:

```
ml /c /coff   files3.asm
link /subsystem:console  files3.obj
```

Программа из листинга 2.6.5 демонстрирует один из возможных алгоритмов обработки текстового файла — построчное чтение текстового файла. Часть программы, занимающаяся чтением и анализом текстового файла, сосредоточена между метками `LOO` и `CONT`. Детально разберитесь в алгоритме и проникнитесь тем, что язык высокого уровня никогда не будет стимулировать написание таких алгоритмов, а значит, язык ассемблера делает нас интеллек-

туально богаче. Обратите внимание, что при выводе строки на консоль (процедура `OUTST`) после каждой ставится символ-разделитель. Для того чтобы строка выводилась на консоль в стандартном текстовом формате, разделитель должен быть символ с кодом `0AH`.

Пример получения временных характеристик файла

Сейчас мы продемонстрируем (см. листинг 2.6.6), как можно получить временные характеристики файлов на основе функции `GetFileTime`, упомянутой ранее в данной главе. Программа выводит имя файла и его временные характеристики создания. Обратите внимание, что имя файла фиксировано и хранится в программе. Переработайте программу так, чтобы имя файла можно было указывать в командной строке.

Листинг 2.6.6. Пример получения временных характеристик файла

```
;files4.asm
.586P
;плоская модель
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
GENERIC_READ      equ 80000000h
GENERIC_WRITE     equ 40000000h
GEN = GENERIC_READ or GENERIC_WRITE
OPEN_EXISTING     equ 3
;прототипы внешних процедур
EXTERN  CreateFileA@28:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  CloseHandle@4:NEAR
EXTERN  GetFileTime@16:NEAR
EXTERN  FileTimeToLocalFileTime@8:NEAR
EXTERN  FileTimeToSystemTime@8:NEAR
EXTERN  wsprintfA:NEAR
;структуры
;структура, выражающая файловое время
FILETIME STRUC
      LOTIME  DD 0
      HITIME  DD 0
```

```
FILETIME ENDS
SYSTIME STRUC
      Y   DW 0   ;год
      M   DW 0   ;месяц
      DWE DW 0   ;день недели
      D   DW 0   ;день месяца
      H   DW 0   ;час
      MI  DW 0   ;минута
      S   DW 0   ;секунда
      MS  DW 0   ;тысячная доля секунды
SYSTIME ENDS
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
      LENS    DD ? ;сюда будет помещена длина строки
      HANDL   DD ? ;здесь дескриптор консоли вывода
      HFILE   DD ? ;дескриптор открываемого файла
      ERRS    DB 'Error!',0 ;сообщение об ошибке
      PATH    DB 'e:\backup3.pst',0 ;путь к файлу
      FTMCR   FILETIME <0> ;для времени создания
      FTMAC   FILETIME <0> ;для времени доступа
      FTMWR   FILETIME <0> ;для времени модификации
      LOCALS1 FILETIME <0> ;для локального времени
      FORM    DB "Write time: Sec %lu Min %lu Hou %lu Day %lu Mon %lu Yea %lu",0
      SST     SYSTIME  <0> ;системный формат времени
      BUF     DB  60 DUP(0);буфер для форматированной строки
      TEXT1   DB  'File: ',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить HANDLE вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;открыть файл
      PUSH 0                 ;должен быть равен 0
      PUSH 0                 ;атрибут файла (если создаем)
      PUSH OPEN_EXISTING     ;как открывать
      PUSH 0                 ;указатель на security attr
      PUSH 0                 ;режим общего доступа
      PUSH GEN               ;режим доступа
      PUSH OFFSET PATH       ;имя файла
      CALL CreateFileA@28
```

```
;проверим, открылся ли файл
      CMP EAX,-1
      JNE CONT
;здесь сообщение об ошибке и выход
      LEA EAX,ERRS
      MOV EDI,1
      CALL WRITE
      JMP EXI
CONT:
      MOV HFILE,EAX
;получить файловое время
      PUSH OFFSET FTMWR
      PUSH OFFSET FTMAC
      PUSH OFFSET FTMCR
      PUSH HFILE
      CALL GetFileTime@16
;преобразовать время к местному
      PUSH OFFSET LOCALS1
      PUSH OFFSET FTMWR
      CALL FileTimeToLocalFileTime@8
;перейти к формату системного времени
      PUSH OFFSET SST
      PUSH OFFSET LOCALS1
      CALL FileTimeToSystemTime@8
;теперь образовать строку
      MOV    AX,SST.Y
      MOVZX  EAX,AX
      PUSH   EAX
      MOV    AX,SST.M
      MOVZX  EAX,AX
      PUSH   EAX
      MOV    AX,SST.D
      MOVZX  EAX,AX
      PUSH   EAX
      MOV    AX,SST.H
      MOVZX  EAX,AX
      PUSH   EAX
      MOV    AX,SST.MI
      MOVZX  EAX,AX
      PUSH   EAX
      MOV    AX,SST.S
      MOVZX  EAX,AX
      PUSH   EAX
      PUSH   OFFSET FORM
      PUSH   OFFSET BUF
      CALL   wsprintfA
```

```
;освободить стек
      ADD    ESP,32
;вывести информацию
      LEA    EAX,TEXT1
      MOV    EDI,0
      CALL   WRITE
      LEA    EAX,PATH
      MOV    EDI,1
      CALL   WRITE
      LEA    EAX,BUF
      MOV    EDI,1
      CALL   WRITE
;закрыть открытый файл
CLOS:
      PUSH   HFILE
      CALL   CloseHandle@4
;выход из программы
EXI:
      PUSH   0
      CALL   ExitProcess@4
;вывести строку (в конце перевод строки)
;EAX - на начало строки
;EDI - с переводом строки или без
WRITE   PROC
;получить длину параметра
      PUSH  EAX
      PUSH  EAX
      CALL  lstrlenA@4
      MOV   ESI,EAX
      POP   EBX
      CMP   EDI,1
      JNE   NO_ENT
;в конце - перевод строки
      MOV  BYTE PTR [EBX+ESI],13
      MOV  BYTE PTR [EBX+ESI+1],10
      MOV  BYTE PTR [EBX+ESI+2],0
      ADD  EAX,2
NO_ENT:
;вывод строки
      PUSH 0
      PUSH OFFSET LENS
      PUSH EAX
      PUSH EBX
      PUSH HANDL
      CALL WriteConsoleA@20
```

```
      RET
WRITE  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.6.6:

```
ml /c /coff   files4.asm
link /subsystem:console  files4.obj
```

В результате выполнения программы из листинга 2.6.6 в текущую консоль будут выведены строка с именем файла и строка, содержащая "секунду минуту час день месяц год" последней модификации файла. В принципе данная программа достаточно проста и комментирования не заслуживает. Обратите только еще раз внимание на использование функции `wsprintfA` — это единственное, что может вызвать какое-либо затруднение у читателя, дошедшего до данной главы.

Глава 2.7

Директивы и макросредства ассемблера

Ну, вот мы добрались и до макросредств, как ни оттягивал я этот момент. Большая часть данной главы будет носить справочный характер. Почему я привожу справочный материал в середине книги, а не в ее начале? Просто я убежден, что справка в начале книги может отбить всякую охоту читать дальше. Многое из того, что я привожу в данной главе, вы уже знаете и, следовательно, с пониманием отнесетесь к материалу этой главы. Кроме этого, речь идет о макросредствах, которые, на мой взгляд, мешают начинающему программисту почувствовать красоту ассемблера. Макросредства обрабатываются на стадии трансляции программы. При помощи них можно управлять трансляцией и придавать программе на ассемблере вид программ на языках высокого уровня.

Метки

Метка с двоеточием после имени определяет адрес следующей за меткой команды. Например:

```
L:
      JMP L
```

Во фрагменте представлена команда `JMP`, которая реализует переход на саму себя.

Директива `LABEL` позволяет определить явно тип метки. Значение же определенной таким образом метки равно адресу команды или данных, стоящих далее. Например, `LABEL L1 DWORD`.

Выражение *`имя`* `PROC` определяет метку, переход на которую обычно происходит по команде `CALL`. Блок кода, начинающийся с такой метки, называют *процедурой*. Впрочем, переход на такую метку можно осуществлять и с помощью `JMP`,

как, впрочем, и команду CALL можно использовать для перехода на обычную метку. В этом, несомненно, состоит сила и гибкость ассемблера.

Директива PROC позволяет указывать параметры для процедуры в стиле языков высокого уровня и локальные переменные. Рассмотрим следующий фрагмент:

```
PROC1 PROC aa:DWORD, bb:DWORD, cc:DWORD
      LOCAL a1:DWORD
      LOCAL a2:DWORD
      MOV   EAX,a2
      ADD   EAX,cc
      RET
PROC1 ENDP
```

В этом фрагменте задается процедура с именем PROC1 с тремя входными параметрами (aa, bb, cc) и двумя локальными переменными (a1, a2). Не правда ли, фрагмент весьма напоминает язык высокого уровня. Вызвать такую процедуру можно также с помощью макрооператора INVOKE PROC1; EAX, EBX, ECX — регистры в директиве играют роль входных параметров. А теперь посмотрим на следующий фрагмент, получившийся в результате обработки транслятором предыдущего:

```
PROC1:
      PUSH EBP
      MOV  EBP,ESP
      ;область для двух локальных переменных
      SUB  ESP,8
      MOV  EAX,DWORD PTR [EBP-8]    ; переменная a2
      ADD  EAX,DWORD PTR [EBP+16]   ; параметр cc
      LEAVE
      RET  12
```

Как видим, для нас это все знакомо и понятно. Имея такой опыт (см., например, листинг 2.6.2 и комментарий к нему), как у вас, можно свободно обходится и без таких макросредств.

В строке за меткой может стоять директива резервирования данных, например: ERR DB "Ошибка" или NUM DWORD 0. С точки зрения языка высокого уровня, таким образом, мы определяем глобальную переменную. С точки же зрения ассемблера нет никакого различия между командой и данными, поэтому между меткой, определяющей команду, и меткой, определяющей данные, нет никакой разницы. Раз уж речь пошла о данных, перечислю их типы: BYTE (DB) — байт, WORD (DW) — 2 байта, DWORD (DD) — 4 байта, FWORD (DF) — 6 байтов, QWORD (DQ) — 8 байтов, TBYTE (DT) — 10 байтов (используется для хранений 80-битовых чисел с плавающей точкой). При резервировании данных можно инициализировать их конкретными значениями: числами и по-

следовательностью символов (байтов). Вместо конкретных числовых значений можно писать символ `?`, который указывает транслятору, что данные не инициализируются. Для резервирования области данных или массива данных используется оператор `DUP`: `S DB 100 DUP(0)` — резервируется сто байтов, инициализированных нулем.

С помощью директивы `EQU` в терминах языков высокого уровня определяются константы. Например: `MES EQU "ERROR!"`, `LAB EQU 145H`. С помощью `EQU` значение данной метке может быть присвоено только один раз. С правой стороны от `EQU` может стоять выражение с использованием арифметических, логических и битовых операций. Вот эти операции: `+`, `-`, `*`, `/`, `MOD` (остаток от деления), `AND`, `OR`, `NOT`, `XOR`, `SHR`, `SHL`. Используются также операции сравнения: `EQ`, `GE`, `GT`, `LE`, `LT`, `NE`. Выражение с операцией сравнения считается логическим и принимает значение 0, если условие не выполняется, и 1 — если выполняется. С помощью директивы `=` можно присваивать только числовые целые значения, но зато производить переприсваивание. Замечу, что выражение может являться операндом команды: `MOV EAX,16*16-1`. Для присвоения чисто строковой константы используются угловые скобки, например: err equ <Неправильный вызов процедуры>.

Метка `$` всегда определяет текущий адрес. Например, рассмотрим следующий фрагмент:

```
START:
      JMP _EXIT
      PUSH OFFSET STROKA
      PUSH OFFSET STROKA
      CALL CharToOemA@8
_EXIT:
      RET
```

В этом фрагменте команду `JMP _EXIT` можно заменить эквивалентной командой `JMP $+17`.

В MASM метки, стоящие в процедуре, автоматически считаются локальными, и, следовательно, имена меток в процедурах могут дублироваться. Не все ассемблеры придерживаются такого подхода. В некоторых, для того чтобы отличать локальные метки, используют специальные префиксы.

Очень часто необходимо, чтобы блок данных в программе начинался на границе, адрес которой был кратен некоторому количеству байтов. Для этой цели используется директива `ALIGN`. После этого ключевого слова идет количество байтов. Например, `ALIGN 4`, `ALIGN 16` и т. д. Ниже представлен фрагмент программы, содержащий сегмент данных, где и используется эта директива:

```
_DATA SEGMENT
      PATH    DB  "C:\1.TXT"
```

```
    HANDLE  DD  ?
    ALIGN   4
    BUF     DB  1000 DUP (0)
    ...
_DATA ENDS
```

Используемая здесь директива ALIGN 4 гарантирует, что буфер, обозначаемый меткой BUF, всегда будет начинаться на границе, кратной 4 байтам.

Строки

Как уже ранее было сказано, при помощи оператора EQU могут быть определены строковые макроконстанты. В арсенале MASM имеется целый набор работы с такими строками.

- Директива CATSTR. Позволяет осуществлять слияние (конкатенацию) нескольких строк.

  ```
  S1  EQU <Hello>
  S2  EQU <World>
  S3  EQU <!>

  MES CATSTR   S1,< >,S2,S3
  ```

  В результате выполнения данного фрагмента константе MES будет присвоена строка "Hello World!".

- Директива INSTR. Данная директива осуществляет поиск подстроки в строке и имеет следующий формат: *name* INSTR [*pos*,] *s1*,*s2*. Здесь *pos* — позиция в строке, откуда начинается поиск, *s1* — строка, где осуществляется поиск, *s2* — подстрока для поиска. Результат поиска (номер символа, откуда начинается найденная подстрока) присваивается *name*. Если подстрока не найдена, то директива возвращает 0 (счет символов в строке идет от 1).
- Директива SUBSTR. Данная директива выделяет подстроку в строке и имеет следующий формат: *name* SUBSTR *s*, *start*, *length*. Здесь *s* — стока, *start* — начальный символ подстроки, *length* — длина подстроки.
- Директива SIZESTR. Данная директива имеет формат: *name* SIZESTR *s* и вычисляет длину строки *s*, присваивая ее *name*.

Структуры

Директива STRUC позволяет объединить несколько разнородных данных в одно целое. Эти данные называются *полями*. Вначале при помощи STRUC определяется шаблон структуры, затем с помощью директивы < > можно определить любое количество экземпляров структур. При этом в угловых скобках

через запятую можно перечислить числовые значения, которые будут присвоены полям. Рассмотрим пример:

```
STRUC COMPLEX
      RE   DD ?
      IM   DD ?
STRUC ENDS
. . .
;в сегменте данных
COMP1 COMPLEX <?>
COMP2 COMPLEX <0> ;инициализация нулем
```

Доступ к полям структуры осуществляется посредством точки, вот так: `MOV COMP1.RE, EAX`. Структуры могут быть вложены друг в друга, и тогда для доступа к полям могут потребоваться две и более точек.

Объединения

Для определения объединения используется директива `UNION`. Объединение очень похоже на структуру, также состоит из отдельных записей. Но есть существенное различие. Длина экземпляра объединения в памяти равна длине самого длинного его поля, при этом все поля будут начинаться по одному адресу — адресу, где будет начинаться и само объединение. Смысл использования объединения заключается в том, что одну и туже область памяти можно рассматривать (трактовать) в контексте различного типа данных.

Пример объединения:

```
EXTCHAR  UNION
      ascii     DB ?
      extascii  DW ?
EXTCHAR  ENDS
```

Данное объединение представляет расширенное определение ASCII-кода. Для использования объединения в программе следует определить объединение, подобно тому, как мы определяли структуры. И далее:

```
;в сегменте данных
easc EXTCHAR <0>
```

В результате мы имеем структуру длиной в два байта. Причем доступ ко всему слову осуществляется через поле `extascii`: `MOV easc.extascii,AX`, а доступ к младшему байту слова через поле `ascii: MOV easc.ascii,AL`.

Удобный прием работы со структурами

Об одном приеме работы со структурами (и объединениями) следует рассказать особо. Основывается он на использовании директивы `ASSUME` (что означает

"примем", "положим"). Вообще в данной книге мы ее почти не используем. При программировании в MS-DOS она использовалась в основном, чтобы указать транслятору, что сегментный регистр указывает на данный сегмент. Однако использовать данную директиву можно не только с сегментными регистрами. Вот фрагмент, который демонстрирует такую технику:

```
COMPLEX STRUC
      RE   DD ?
      IM   DD ?
COMPLEX ENDS
.
.
.
;в сегменте данных
COMP COMPLEX <0>
.
.
.
;в сегменте кода
MOV EBX,OFFSET COMP
ASSUME EBX:PTR COMPLEX
MOV    [EBX].RE,10
MOV    [EBX].IM,10
ASSUME EBX:NOTHING
```

В действительности команда `MOV [EBX].RE,10` эквивалентна просто `MOV DWORD PTR [EBX],10`, а команда `MOV [EBX].IM,10` эквивалентна `MOV DWORD PTR [EBX+4],10`. Согласитесь, что это удобно.

Условное ассемблирование

Условное ассемблирование дает возможность при трансляции обходить тот или иной участок программы. Существуют три вида условного ассемблирования:

- первый:

```
IF выражение
...
ENDIF
```

- второй:

```
IF выражение
...
ELSE
...
ENDIF
```

- третий:

  ```
  IF выражение1
  ...
  ELSEIF выражение2
  ...
  ELSEIF выражение3
  ...
  ELSE
  ...
  ENDIF
  ```

Условие считается невыполненным, если выражение принимает значение 0, и выполненным, если выражение отлично от нуля.

Ассемблер MASM поддерживают также несколько условных специальных директив, назовем некоторые из них.

- ```
  IFE выражение
  ...
  ELSEIFE
  ...
  ENDIFE
  ```

- Операторы `IF1` и `IF2` проверяют первый и второй проход при ассемблировании.
- Оператор `IFDEF` проверяет, определено ли в программе символическое имя, `IFDEFN` — обратный оператор.

Имеются и другие `IF`-операторы. Описание их есть в любом справочнике по ассемблеру.

Существует целый набор директив, начинающихся с `.ERR`. Например, `.ERRE` *`выражение`* вызовет прекращение трансляции и сообщение об ошибке, если выражение станет равным 0.

Условное ассемблирование понадобится нам в конце главы для написания программы, транслируемой одновременно в MASM и TASM.

Вызов процедур

С упрощенным вызовом процедур в MASM вы уже познакомились. Это директива `INVOKE`. Процедура должна быть заранее определена с использованием ключевого слова `PROTO` (от англ. *prototype* — прототип). Например:

```
MessageBoxA PROTO :DWORD,:DWORD,:DWORD,:DWORD
...
;и далее вызов
invoke MessageBox, h, ADDR TheMsg, ADDR TitleW, MB_OK
```
```

Здесь `h` — дескриптор окна, откуда вызывается сообщение, `TheMsg` — строка сообщения, `TitleW` – заголовок окна, `MB_OK` — тип сообщения. `ADDR` в данном случае — синоним `OFFSET`.

Макроповторения

Существуют разные способы макроповторений.

- Повторение, заданное определенное число раз. Используется макродиректива `REPT`.

  Например:

  ```
  A EQU 10
  REPT 100
  DB A
  ENDM
  ```

  Будет сгенерировано 100 директив `DB 10`. С этой директивой удобно использовать оператор `=`, который позволяет изменять значение переменной многократно, т. е. использовать выражение типа `A = A + 5`.

- Директива `IRP`.

  ```
  IRP параметр,<список>
  ...
  ENDM
  ```

  Блок будет вызываться столько раз, сколько параметров в списке. Например:

  ```
  IRP REG,<EAX,EBX,ECX,EDX,ESI,EDI>
    PUSH REG
  ENDM
  ```

  приведет к генерации следующих строк:

  ```
  PUSH EAX
  PUSH EBX
  PUSH ECX
  PUSH EDX
  PUSH ESI
  PUSH EDI
  ```

- Директива `IRPC`.

  ```
  IRPC параметр, строка
     Операторы
  ENDM
  ```

  Пример:

  ```
  IRPC CHAR,azklg
     CMP  AL,'&CHAR&'
  ```

```
   JZ  EndC
ENDM
EndC:
```

Данный фрагмент эквивалентен такой последовательности:

```
CMP AL, 'a'
JZ  EndC
CMP AL, 'z'
JZ  EndC
CMP AL, 'k'
JZ  EndC
CMP AL, 'l'
JZ  EndC
CMP AL, 'g'
JZ  EndC
EndC:
```

Амперсанд (&) в последнем примере используется для того, чтобы задать вычисление параметра блока повторения даже внутри кавычек. Амперсанд — это макрооперация, которая работает в блоке повторения, поскольку блоки повторения представляют собой один из типов макрокоманды.

Макроопределения

Общий вид макроопределения:

```
Имя MACRO параметры
.
.
.
ENDM
```

Определив блок один раз, можно использовать его в программе многократно. Причем в зависимости от значений параметров заменяемый участок может иметь разные значения. Если заданный участок предполагается многократно использовать, например в цикле, макроопределение имеет несомненные преимущества перед процедурой, т. к. несколько ускоряет выполнение кода. Вызов процедуры предполагает, по крайней мере, две дополнительные команды: `CALL` и `RET`. Пример:

```
EXC MACRO par1, par2
     PUSH  par1
     POP   par2
ENDM
```

Данное макроопределение приводит к обмену содержимым между параметрами.

Строка кода

```
EXC EAX,EBX
```

эквивалентна `PUSH EAX` / `POP EAX`, а строка

```
EXC MEM1,ESI
```

эквивалентна `PUSH MEM1` / `POP ESI` и т. д. Замечу, что если первый параметр будет непосредственно числом, то это приведет к загрузке данного числа во второй операнд.

Важным вопросом в связи с макроопределениями является проблема меток. Действительно, если мы будем применять в макроопределении обычные метки, то при использовании его более чем один раз возникнет коллизия (метка с одним и тем же именем повторится в тексте программы). Коллизия эта разрешается при помощи объявления локальных меток. Для этого используется ключевое слово `LOCAL`. Например:

```
EXC MACRO par1,par2
      LOCAL EXI
      CMP   par1,par2
      JE    EXI
      PUSH  par1
      POP   par2
EXI:
ENDM
```

Данное макроопределение можно использовать сколь угодно много раз — при каждой подстановке ассемблер будет генерировать уникальную метку.

Для выхода из макроопределения (т. е. для прекращения генерации макроопределения) применяется директива `EXITM`. Она может понадобиться, если в макроопределении вы используете условные конструкции типа `IF..ENDIF`.

Приведу пример еще одного весьма полезного макроса:

```
ustring MACRO quoted_text, ptr_buf
LOCAL asc_txt
.data
      asc_txt db quoted_text,0
.code
      invoke MultiByteToWideChar,CP_ACP,0,
             OFFSET asc_txt,-1,OFFSET ptr_buf,LENGTHOF ptr_buf
ENDM
```

Макрос преобразует указанную строку в кодировке ASCII в строку с кодировкой Unicode и помещает ее в буфер, на который указывает переменная `ptr_buf`. В более привычном для нас виде макрос будет выглядеть так:

```
ustring MACRO quoted_text, ptr_buf
LOCAL asc_txt
.data
      asc_txt db quoted_text,0
```

```
.code
      PUSH LENGTHOF ptr_buf
      PUSH OFFSET BUF      ;адрес буфера
      PUSH -1
      PUSH OFFSET asc_txt
      PUSH 0
      PUSH 0
      CALL MultiByteToWideChar@24
ENDM
```

Взгляните, например, на следующий фрагмент программы:

```
ustring "Привет! ",buf ;buf — в сегменте данных
PUSH 0
PUSH OFFSET buf
PUSH OFFSET buf
PUSH 0
CALL MessageBoxW@16 ;вывод строки в кодировке Unicode
```

Не правда ли, удобно?!

Некоторые другие директивы транслятора ассемблера

Еще несколько полезных директив MASM.

- Кроме объявлений с использованием директив `PUBLIC` и `EXTERN`, возможно объявление при помощи директивы `GLOBAL`, которая действует, как `PUBLIC` и `EXTERN` одновременно.
- `PURGE` *имя макроса*. Отменяет загрузку макроса. Используется при работе с библиотекой макросов, чтобы не перегружать память[1].
- `LENGTHOF` — определяет число элементов данных. `SIZEOF` — определяет размер данных.
- Директивы задания набора команд:
    - `.8086` — разрешены только команды микропроцессора 8086. Данная директива работает по умолчанию;
    - `.186` — разрешены команды 186;
    - `.286` и `.286P` — разрешены команды 286-го микропроцессора. Добавка "P" здесь и далее означает разрешение команд защищенного режима;
    - `.386` и `.386P` — разрешение команд 386-го микропроцессора;
    - `.486` и `.486P` — разрешение команд 486-го процессора;
    - `.586` и `.586P` — разрешены команды P5 (Pentium);

---

[1] В операционной системе MS-DOS это было существенно.

- `.686` и `.686P` — разрешены команды P6 (Pentium Pro, Pentium II);
- `.8087` — разрешены команды арифметического сопроцессора 8087;
- `.287` — разрешены команды арифметического сопроцессора 287;
- `.387` — разрешены команды арифметического сопроцессора 387;
- `.MMX` — разрешены команды расширения MMX.

❒ Директивы управления листингом:

- `NAME` — задать имя модуля;
- `TITLE` — определяет заголовок листинга;

Замечание

По умолчанию и имя модуля, и заголовок листинга совпадают с именем файла, где хранится программа.

- `SUBTTL` — определяет подзаголовок листинга;
- `PAGE` — определяет размеры страницы листинга: длина, ширина. Директива `PAGE` без аргументов начинает новую страницу листинга;
- `.LIST` — выдавать листинг;
- `.XLIST` — запретить выдачу листингу;
- `.SALL` — подавить печать макроопределений;
- `.SFCOND` — подавить печать условных блоков с ложными условиями;
- `.LFCOND` — печатать условные блоки с ложными условиями;
- `.CREF` — разрешить листинг перекрестных ссылок;
- `.XCREF` — запретить листинг перекрестных ссылок.

Конструкции времени исполнения программы

Перечисленные далее конструкции преобразуются при ассемблировании в команды микропроцессора. Это уже прямой выход к возможностям языков высокого уровня.

❒ *Условные конструкции.*

- ```
  .IF условие
    ...
  .ENDIF
  ```
- ```
  .IF условие
    ...
  .ELSE
  ```

```
  ...
.ENDIF
```

- ```
  .IF условие1
    ...
  .ELSEIF условие2
    ...
  .ELSEIF условие3
    ...
    ...
  .ELSE
    ...
  .ENDIF
  ```

Рассмотрим следующий фрагмент, содержащий условную конструкцию, и соответствующий ей ассемблерный код:

```
.IF EAX==12H
    MOV EAX,10H
.ELSE
    MOV EAX,15H
.ENDIF
```

Представленный выше фрагмент эквивалентен следующему ассемблерному коду:

```
      CMP EAX,12H
      JNE NO_EQ
      MOV EAX,10H
      JMP EX_BLOK
NO_EQ:
      MOV EAX,15H
EX_BLOK:
```

Весьма удобная штука, но не увлекайтесь: на мой взгляд, это сильно расслабляет, в конце концов, вы просто забудете некоторые команды процессора, будете писать на "жутком" диалекте: смеси команд процессора и высокоуровневых операторов[2].

- *Цикл "пока".*

```
.WHILE условие
 ...
.ENDW
```

Пример:

```
WHILE EAX<64H
  ADD EAX,10H
ENDW
```

[2] Смесь французского с нижегородским.
```

Вот как этот фрагмент будет записан обычными командами процессора:

```
    JMP L2
L1:
    ADD EAX,10H
L2:
    CMP EAX,64H
    JB L1
```

Пример программы одинаково транслируемой как в MASM, так и в TASM

Написание программы, которую можно транслировать без изменений различными ассемблерами, мне кажется довольно актуально, ведь не всегда под рукой оказывается нужный ассемблер. В качестве альтернативного ассемблера логично было бы выбрать ассемблер TASM, когда-то очень популярный и до сих пор часто используемый ассемблерщиками всего мира. Имя TASM — это сокращение Turbo Assembler, т. е. турбо ассемблер. Когда-то TASM интенсивно разрабатывался фирмой Borland, которая некоторое время назад почему-то охладела к своему продукту.

> **Замечание**
>
> В предыдущих изданиях данной книги я довольно подробно рассматриваю вопрос о программировании на TASM. Однако в связи с тем, что данный ассемблер больше не поддерживается Borland, я также изъял материалы о TASM из своих книг.

Для создания программы, одинаково транслируемой двумя ассемблерами, прекрасно подходят операторы условного ассемблирования. Удобнее всего использовать `IFDEF` и возможности трансляторов задавать символьную константу, все равно — TASM или MASM. И в ML, и в TASM32 определен ключ `/D`, позволяющий задавать такую константу.

В листинге 2.7.1 представлена программа, транслируемая и в MASM, и TASM. Программа весьма проста, но рассмотрения ее вполне достаточно для создания более сложных подобных совместимых программ.

Листинг 2.7.1. Пример использования условного ассемблирования для написания совместимой программы

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;проверить, определена символьная константа MASM или нет
IFDEF MASM
;работаем в MASM
```

```
      EXTERN      ExitProcess@4:NEAR
      EXTERN      MessageBoxA@16:NEAR
      includelib c:\masm32\lib\kernel32.lib
      includelib c:\masm32\lib\user32.lib
ELSE
;работаем в TASM
      EXTERN      ExitProcess:NEAR
      EXTERN      MessageBoxA:NEAR
      includelib c:\tasm32\lib\import32.lib
      ExitProcess@4 = ExitProcess
      MessageBoxA@16 = MessageBoxA
ENDIF
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG DB "Простая программа",0
      TIT DB "Заголовок",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
      PUSH 0
      PUSH OFFSET TIT
      PUSH OFFSET MSG
      PUSH 0 ;дескриптор экрана
      CALL MessageBoxA@16
;---------------------------------------
      PUSH 0
      CALL ExitProcess@4
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.7.1:

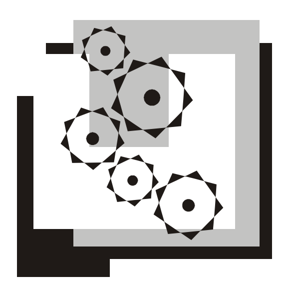
- трансляция в MASM

```
ML /c /coff /DMASM PROG.ASM
LINK /SUBSYSTEM:WINDOWS PROG.OBJ
```

- трансляция в TASM

```
TASM32 /ml PROG.ASM
TLINK32 -aa PROG.OBJ
```

Как видите, все сводится к проверке, определена символьная константа MASM или нет (ключ `/DMASM`). Еще одна сложность — добавка в конце имени `@N`. Эту проблему мы обходим, используя оператор `=`, с помощью которого переопределяем имена (см. секцию "работаем в TASM").



Глава 2.8

Еще об управлении файлами (*CreateFile* и другие функции)

В силу значимости функции `CreateFile` я решил посвятить ей отдельную главу, где попытаюсь показать все многообразие возможностей этой многоликой функции. Здесь же я расскажу о некоторых других функциях API, связанных с управлением файлами. Кроме файлов, разговор в данной главе пойдет и о других устройствах ввода/вывода.

Полное описание функции *CreateFile* для работы с файлами

В операционной системе Windows используется такое понятие, как "устройство". Это позволяет унифицировать передачу и получение информации, т. е. использовать для передачи и получения данных одни и те же функции API. В табл. 2.8.1 перечислены эти устройства.

Таблица 2.8.1. *Основные устройства обмена информацией в операционной системе Windows*

Название устройства	Пояснение
Каналы (pipes)	Существуют два вида каналов: именованные и анонимные. Именованные каналы используются для связи источника и получателя через локальную сеть. Анонимные каналы выполняют аналогичную функцию, но в рамках одного компьютера
Каталоги (Directories)	Существуют как структурная единица файловых систем. Обмен информации непосредственно с каталогом может означать изменение его атрибутов, например атрибута сжатия файлов, содержащихся в каталоге

Таблица 2.8.1 *(окончание)*

Название устройства	**Пояснение**
Коммуникационные ресурсы (Communications Resource)	В основном это относится к портам COM и LPT, служащим для обмена информацией с такими устройствами, как принтер, модем и т. п.
Консоль (console)	Текстовый экран. Ввод/вывод на текстовый экран
Логический диск (Logical device)	Раздел жесткого диска, накопитель на гибком диске. Основной операцией, которая может вызвать обмен данных, является форматирование
Почтовый ящик (mailslot)	Позволяет передавать информацию от нескольких источников одному получателю в пределах одного компьютера, локальной компьютерной сети или домена[1]
Сокет (socket)	Является устройством обмена информацией между двумя машинами, поддерживающими механизм сокетов
Файл (File)	Устройство для долговременного хранения больших объемов информации
Дополнительный поток файла (Stream)	Файл кроме основного потока данных может содержать и другие потоки (см. описание NTFS в *главе 2.6*), содержащие дополнительную информацию о файле. В листинге 2.8.4 приведен пример использования еще одного потока для файла
Физический диск (Physical device)	Доступ к структурам жесткого диска (таблица разделов, структуры, расположенные на разделе)

А теперь, после того как вы рассмотрели содержимое табл. 2.8.1, я сообщу вам удивительную вещь: большую часть перечисленных устройств можно открыть с помощью функции `CreateFile`. Теперь ясно, что данная функция заслуживает самого пристального внимания.

В данном разделе мы рассмотрим функцию в контексте управления файлами. Поскольку первым аргументом функции `CreateFile` является адрес полного имени файла (путь к файлу и его имя), рассмотрим вкратце правила, которыми следует руководствоваться при формировании этой строки.

- Для формирования полного имени файла используются символы текущей кодовой страницы с кодами, большими 31.
- Для отделения компонентов в имени используются символы `/` и `\`. Полное имя файла на общем сетевом ресурсе должно начинаться с `\\server\share`.

[1] Домен — группа рабочих станций и серверов локальной сети Microsoft, объединенных общей базой данных сетевых ресурсов.


Здесь server — имя сетевого компьютера, share — имя сетевого ресурса (согласно UNC — Universal Naming Convention, т. е. универсальному соглашению об именах).

- Символ . обозначает текущий каталог, .. — родительский каталог. Кроме этого, последний в строке символ . отделяет от имени так называемое расширение, используемое операционной системой для распознавания типа файла.
- Для формирования полного имени файла и каталога нельзя использовать символы <, >, :, /, |, \.
- Полное имя файла должно быть представлено строкой, оканчивающейся нулем. Максимальная длина такой строки определяется специальной системной константой MAX_LEN, которая в настоящее время равна 260. Для использования более длинных строк следует воспользоваться кодировкой Unicode и соответствующей версией функции CreateFile (т. е. с суффиксом W). В этом случае строка должна начинаться с префикса \\?\. Строка при этом может достигать длины 32 000 символов!
- Функции, использующие имена файлов в качестве параметров, не чувствительны к регистрам английских букв.
- В качестве имен файлов и каталогов нельзя использовать зарезервированные слова: CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8 и LPT9.

> **Замечание**
>
> Перечисленные в табл. 2.8.1 устройства на деле являются представителями класса объектов ядра. Точнее, таковыми они становятся после того, создаются с помощью функции CreateFile или другой подобной функции (название CreateFile, как видите, правильно отражает суть дела). Структуры, описывающие объекты, хранятся в области ядра, т. е. защищены от доступа со стороны пользовательских программ. Весь доступ к таким объектам осуществляется посредством дескриптора, возвращаемого при создании объекта. Замечу при этом, что дескриптор описывает объект лишь в пределах данного процесса.

Перейдем к рассмотрению параметров функции CreateFile.

- 1-й параметр — адрес строки, содержащей имя создаваемого (или открываемого) файла. Строка должна заканчиваться нулем.
- 2-й параметр определяет, как будет осуществляться обмен данными с устройством. Возможны следующие значения:
  - 0 — предполагается, что запись или чтение данных производиться не будет. Можно производить лишь изменение параметров файла (временные характеристики, атрибут и т. п.);

- `GENERIC_READ = 80000000h` — предполагается чтение из файла;
- `GENERIC_WRITE = 40000000h` — предполагается запись в файл;
- комбинация констант `GENERIC_READ | GENERIC_WRITE` разрешает как чтение, так и запись в файл.

Кроме этого параметр может содержать ряд других флагов, конкретизирующих права доступа к данному файлу или другому устройству. Все они перечислены в документации и используются достаточно редко. Примером может служить значение `DELETE = 10000h`, предполагающее возможность удаления файла (объекта).

- ❒ 3-й параметр определяет тип доступа к файлу. Можно сказать, желаемый доступ, т. к. реально доступ может быть ограничен тем, что файл может быть уже открыт другим процессом. Возможные значения параметра:
  - 0 — требование, чтобы другие процессы не могли иметь доступ к данному файлу (монопольное владение). В свою очередь, если файл уже открыт другим процессом, то данное значение параметра не позволит вам открыть файл;
  - `FILE_SHARE_READ = 1` — требование, чтобы другие процессы не могли записывать на это устройство. Если файл уже открыт для записи, то вы не сможете его открыть;
  - `FILE_SHARE_WRITE = 2` — требование, чтобы другие процессы не могли читать данный файл. Если файл уже открыт для чтения, то вы не сможете его открыть;
  - значение `FILE_SHARE_WRITE | FILE_SHARE_READ`. Вы допускаете чтение и запись в открываемый вами файл другими процессами. Открытие будет неудачным только при монопольном владении данным файлом другим процессом;
  - значение `FILE_SHARE_DELETE = 4`. Допускается доступ других процессов для удаления файла (объекта).
- ❒ 4-й параметр указывает на специальную структуру `SECURITY_ATTRIBUTES` (дескриптор защиты или безопасности). Данная структура позволяет задать информацию о защите и определить, будет ли наследоваться дескриптор, который возвращает функция `CreateFile`. Чаще всего данный параметр полагается равным `NULL` (т. е. 0). Это означает, что дескриптор не наследуется. Рассмотрим структуру `SECURITY_ATTRIBUTES`.

```
SECURITY_ATTRIBUTES STRUC
      L     DD ?
      DESC  DD ?
      INHER DD ?
SECURITY_ATTRIBUTES ENDS
```

Как видим, структура состоит всего из трех полей. Первое поле определяет длину всей структуры, т. е. в данном случае должно составлять 12. Второе поле структуры — это наследуемый дескриптор. Наконец, третье поле принимает значение 0 или 1. При значении 1 дочерние процессы будут наследовать дескриптор.

- 5-й параметр определяет поведение функции `CreateFile` в случае наличия или отсутствия файла с указанным именем:
  - `CREATE_NEW = 1` — предписывает создавать новый файл, если файл с указанным именем отсутствует, в противном случае функция не выполняется;
  - `CREATE_ALWAYS = 2` — предписывает создавать файл в любом случае. Если файл уже существует, то он переписывается и его длина становится равной 0;
  - `OPEN_EXISTING = 3` — предписывает открывать файл, если он существует;
  - `OPEN_ALWAYS = 4` — предписывает открыть файл, если он существует, или создать файл, если файл отсутствует;
  - `TRUNCATE_EXISTING = 5` — предписывает открыть существующий файл и обнулить его размер. Если файл отсутствует, то функция не выполняется (*ср.* `CREATE_ALWAYS`).
- 6-й параметр используется в основном для определения атрибутов создаваемого файла (см. список атрибутов в *главе 2.5*). Кроме этого, здесь же можно указать флаги, позволяющие системе оптимизировать алгоритмы кэширования. Есть и другие флаги. Вот они:
  - `FILE_FLAG_NO_BUFFERING = 20000000h` — доступ к файлу должен осуществляться без буферизации данных;
  - `FILE_FLAG_SEQUENTIAL_SCAN = 8000000h` — при установке этого флага система полагает, что осуществляется последовательный доступ к файлу. Соответственно при последовательном чтении может быть достигнута максимальная скорость считывания;
  - `FILE_FLAG_RANDOM_ACCESS = 10000000h` — данный флаг используется для указания, что система не должна считывать слишком много лишних данных. Его следует использовать, если предполагается частое позиционирование в файле;
  - `FILE_FLAG_WRITE_THROUGH = 80000000h` — флаг запрещает промежуточное кэширование при записи в файл. При этом все изменения записываются сразу на диск;

- `FILE_FLAG_DELETE_ON_CLOSE = 4000000h` — если установлен этот флаг, то операционная система удаляет этот файл после закрытия всех его дескрипторов;
- `FILE_FLAG_BACKUP_SEMANTICS = 2000000h` — данный флаг используется в программах резервного копирования;
- `FILE_FLAG_POSIX_SEMANTICS = 1000000h` — заставляет систему учитывать регистр букв в имени при создании и открытии файла;
- `FILE_FLAG_OPEN_REPARSE_POINT = 200000h` — данный флаг предписывает системе игнорировать наличие у системы точки повторной обработки *(см. главу 2.6)*;
- `FILE_FLAG_OPEN_NO_RECALL = 100000h` — если флаг установлен, то система не восстанавливает файл из хранилища *(см. главу 2.6)*;
- `FILE_FLAG_OVERLAPED = 40000000h` — флаг задает асинхронный обмен данными с устройством. Об асинхронном вводе/выводе мы поведем разговор несколько позже.

❒ 7-й параметр может содержать дескриптор уже открытого файла. В этом случае при открытии уже существующего файла используются атрибуты, определяемые данным параметром. Обычно этот параметр задают равным значению `NULL` (0).

При неудачном выполнении функция возвращает значение `INVALID_HANDLE_VALUE = -1`.

Замечание

Начинающим программистам лишний раз хочу указать, что термином "файл" называется и объект файловой системы, хранящийся во внешней памяти, и объект ядра операционной системы, создаваемый функцией `CreateFile` и связанный с файлом в файловой системе. Таких терминологических двойников пока, к сожалению, много в разных разделах быстро развивающейся науки "Информатика" (см., например, почтовый ящик mailslot, не имеющий никакого отношения к электронной почте).

Другие возможности функции *CreateFile*

Функция `CreateFile` — действительно очень универсальная функция. Впрочем, это всего лишь отражение концепции устройства, принятого в операционной системе Windows. В листинге 2.6.4 мы уже использовали функцию `CreateFile` для открытия вывода на консоль. Это всего лишь один пример. В качестве другого примера приведу работу с таким устройством, как почтовый ящик.

Почтовый ящик или mailslot

Этот почтовый ящик еще называют mailslot. Данное устройство позволяет осуществлять обмен информацией между процессами в рамках не только одного компьютера, но и в рамках локальной компьютерной сети. Беда лишь в том, что объем почтового ящика не превышает 64 Кбайт. Впрочем, узнав механизм передачи данных с помощью устройства mailslot, вы поймете, что размер почтового ящика в принципе не важен, т. к. при двухсторонней связи можно наладить передачу информации последовательными порциями.

Суть использования почтового ящика заключается в следующем.

- Любой процесс может создать почтовый ящик с помощью функции `CreateMailslot`. При удачном создании процесс получает дескриптор почтового ящика и возможность читать из него с помощью функции `ReadFile` (как будто это файл). Процесс, создавший почтовый ящик, называется сервером.
- Если почтовый ящик создан, то любой процесс может подсоединиться к нему (открыть почтовый ящик) с помощью функции `CreateFile` (!) и записать туда порцию информации с помощью обычной функции `WriteFile`.
- Ящик закрывается при уничтожении процесса, его породившего, или закрытии всех дубликатов дескриптора ящика с помощью функции `CloseHandle`.
- При работе с почтовым ящиком следует использовать такие соглашения об имени почтового ящика.
  - При создании почтового ящика в общем случае используется имя `\\.\mailslot\[path]name`. Здесь *`name`* — имя почтового ящика, *`path`* — путь, который может состоять из нескольких каталогов, отделенных обратной косой чертой. Эти каталоги не имеют ничего общего с реально существующими каталогами на диске.
  - При открытии почтового ящика для записи туда используется то же имя, что и при создании. Например, если был создан почтовый ящик `texts`, т. е. использовалось полное имя `\\.\mailslot\texts`, то и при открытии ящика для записи должна использоваться такая же строка.
  - Если необходимо открыть почтовый ящик для записи, который находится на другом компьютере локальной сети, то используется следующая строка: `\\ComputerName\mailslot\[path]name`. Здесь *`ComputerName`* — сетевое имя компьютера.
  - Интересно, что если процессы, создающие почтовые ящики, функционируют в одном домене, то можно создать коллективный ящик для

нескольких процессов. Для этого все процессы должны создать ящик с одним и тем же именем. Если теперь с помощью функции `CreateFile` открыть почтовый ящик для записи с именем вида `\\`*`DomainName`*`\mailslot\[`*`path`*`]`*`name`*, где *`DomainNaim`* — имя домена, то сообщения будут получать все процессы, создававшие этот ящик. Кроме этого можно использовать также имя `\\*\mailslot\[`*`path`*`]`*`name`* для главного домена.

Итак, теория о почтовых ящиках изложена, и можно приступать к реализации конкретных примеров. В листингах 2.8.1 и 2.8.2 представлены тексты программ сервера, создающего почтовый ящик и читающего оттуда, и клиента, открывающего почтовый ящик и посылающего туда сообщения.

Листинг 2.8.1. Программа-сервер (server.asm) создает почтовый ящик и ожидает прихода сообщения

```
;сервер, создающий и читающий из mailslot
.586P
;плоская модель
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE     equ -11
MAILSLOT_WAIT_FOREVER equ -1
;прототипы внешних процедур
EXTERN  ReadFile@20:NEAR
EXTERN  CloseHandle@4:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  CreateMailslotA@16:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetStdHandle@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      LENS  DD ? ;сюда будет помещена длина строки
      PATHM DB "\\.\mailslot\mail1",0
      BUFER DB 1000 DUP(0)
      H     DD ?
      N     DD ?
      HANDL DD ?
      ERRS  DB 'Error!',0
_DATA ENDS
;сегмент кода
```

```
_TEXT SEGMENT
START:
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;создать почтовый ящик
      PUSH 0
      PUSH MAILSLOT_WAIT_FOREVER
      PUSH 0
      PUSH OFFSET PATHM
      CALL CreateMailslotA@16
      CMP  EAX,-1
      JNZ  CON
      LEA  EAX,ERRS
      MOV  EDI,1
      CALL WRITE
      JMP  EXI
CON:
      MOV  H,EAX
;читать из почтового ящика
      PUSH 0
      PUSH OFFSET N
      PUSH 1000
      PUSH OFFSET BUFER
      PUSH H
      CALL ReadFile@20
;вывести содержимое
      LEA EAX,BUFER
      MOV EDI,1
      CALL WRITE
;закрыть почтовый ящик
      PUSH H
      CALL CloseHandle@4
;выйти из программы
EXI:
      PUSH 0
      CALL ExitProcess@4
;вывести строку (в конце перевод строки)
;EAX - на начало строки
;EDI - с переводом строки или без
WRITE   PROC
;получить длину параметра
        PUSH  EAX
        PUSH  EAX
        CALL  lstrlenA@4
        MOV   ESI,EAX
```

```
        POP   EBX
        CMP   EDI,1
        JNE   NO_ENT
;в конце - перевод строки
        MOV  BYTE PTR [EBX+ESI],13
        MOV  BYTE PTR [EBX+ESI+1],10
        MOV  BYTE PTR [EBX+ESI+2],0
        ADD  EAX,2
NO_ENT:
;вывод строки
        PUSH 0
        PUSH OFFSET LENS
        PUSH EAX
        PUSH EBX
        PUSH HANDL
        CALL WriteConsoleA@20
        RET
WRITE ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.8.1:

```
ML /c /coff server.ASM
LINK /SUBSYSTEM:CONSOLE server.OBJ
```

Программа server.asm создает почтовый ящик и далее вызывает функцию `ReadFile`. Обратите внимание, что при создании почтового ящика мы установили параметр `MAILSLOT_WAIT_FOREVER`, что означает, что функция `ReadFile` будет бесконечно долго ждать, когда в ящике появятся данные. Функция возвращает управление только тогда, когда данные появятся. Содержимое ящика затем выводится на консоль.

Листинг 2.8.2. Программа-клиент (klient.asm) открывает почтовый ящик и записывает туда информацию

```
;клиент, открывающий mailslot и пишущий туда
.586P
;плоская модель
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
GENERIC_WRITE     equ 40000000h
FILE_SHARE_READ   equ 1h
OPEN_EXISTING     equ 3
;прототипы внешних процедур
EXTERN  WriteFile@20:NEAR
```

```
EXTERN  CreateFileA@28:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  CloseHandle@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;----------------------------------------------
;сегмент данных
_DATA SEGMENT
      PATHM DB "\\.\mailslot\mail1",0
      H     DD ?
      MES   DB 'Hello! Server!',0
      N     DD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;открыть почтовый ящик
      PUSH 0                  ;должен быть равен 0
      PUSH 0                  ;атрибут файла - не имеет значения
      PUSH OPEN_EXISTING      ;как открывать
      PUSH 0                  ;указатель на security_attr
      PUSH FILE_SHARE_READ    ;режим общего доступа
      PUSH GENERIC_WRITE      ;режим доступа
      PUSH OFFSET PATHM       ;имя почтового ящика
      CALL CreateFileA@28
      MOV  H,EAX
;записать в него данные
      PUSH 0
      PUSH OFFSET N
      PUSH 16 ;длина сообщения
      PUSH OFFSET MES
      PUSH H
      CALL WriteFile@20
;закрыть почтовый ящик
      PUSH H
      CALL CloseHandle@4
;выход
EXI:
      PUSH 0
      CALL ExitProcess@4
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.8.2:

```
ML /c /coff klient.ASM
LINK /SUBSYSTEM:CONSOLE klient.OBJ
```

Программа-клиент (см. листинг 2.8.2) открывает почтовый ящик посредством функции `CreateFile`. С помощью полученного дескриптора можно отправлять в ящик данные. В программе отправляется строка (вместе с 0 на конце).

Представленные выше программы очень просты. Программа-сервер осуществляет одноразовое чтение из почтового ящика. В действительности нет никакого труда в том, чтобы использовать ящик для постоянного обмена информацией. Для этого надо помнить о двух вещах:

- при чтении из ящика он освобождается;
- при записи в ящик его содержимое заменяется новым.

В принципе клиент также может создать ящик, и тогда связь между клиентом и сервером будет двунаправленной (дуплексной).

Каналы передачи информации (pipes)

Каналы являются эффективным способом двустороннего обмена данными между процессами. Существуют два вида каналов: *анонимные* и *именованные*. Анонимные каналы мы будем разбирать в *главе 3.5* (см. листинг 3.5.4). Этот вид канала удобно использовать в рамках одного приложения для обмена информацией между двумя процессами — родительским и дочерним. Именованные каналы — вещь более мощная. Они позволяют обмениваться данными приложениям, находящимся на разных компьютерах в локальной сети. В частности MS SQL Server в качестве одного из механизмов обмена информацией с клиентами предлагает как раз именованные каналы.

Для того чтобы именованный канал заработал, он должен быть создан. Для этого используется функция `CreateNamePipe`. Процесс, создающий канал, называется сервером. При создании именованного канала задаются некоторые параметры, влияющие на то, как в дальнейшем будет функционировать данный объект. К таким параметрам относится, в частности, параметр, определяющий режим взаимодействия через канал. Можно задать три режима взаимодействия:

- двусторонний (дуплексный). Данный метод предполагает возможность движения информации по каналу в обе стороны — от сервера к клиенту и обратно;
- два односторонних метода. Предполагают движение информации только в одну сторону.

В общем виде имя канала представляется следующей строкой: `\\.\pipe\`*`pipename`*. Здесь *`pipename`* — имя канала. Как и в случае с почтовым ящиком (mailslot), можно задать режим бесконечного ожидания для данного

канала. В этом случае функции ReadFile и WriteFile закончат выполнение, только когда прекратится передача данных. После создания канала следует использовать функцию ConnectNamedPipe, чтобы разрешить клиентским процессам подключиться к каналу, другими словами, перевести процесс-сервер в состояние ожидания.

Клиентский процесс, как и в случае почтового ящика, может подключиться к каналу с помощью всеобъемлющей функции CreateFile. Для открытия он должен использовать ту же структуру имени для канала, т. е. \\.\pipe*pipename*. Клиентский процесс, таким образом, должен заранее знать имя канала, к которому он подключается. Если попытка окончилась неудачей, то следует проверить с помощью функции GetLastError, в чем причина ошибки. Если функция возвратит ERROR_PIPE_BUSY = 231[2], то это означает, что канал занят другим процессом и следует подождать. Для ожидания может быть использована функция WaitNamedPipe.

Дисковые устройства

Функция CreateFile может открывать и дисковые устройства. Для открытия первого диска вашего компьютера следует использовать имя \\.\PhysicalDrive0, а для открытия логического раздела C: — имя \\.\C:. Но самое замечательное здесь то, что после открытия вы можете использовать функции ReadFile и WriteFile[3], т. е. читать и писать непосредственно кластеры и сектора. Кроме этого, в вашем распоряжении имеется еще функция DeviceIoControl *(см. подробнее главу 4.6)*, которая может выполнять самые разные операции, в том числе получение статистической информации о диске или даже (о боже!) форматирование. На ней мы останавливаться не будем, а рассмотрим простой пример чтения главной загрузочной записи диска. При открытии устройства она (т. е. загрузочная запись), естественно, оказывается в начале нашего гипотетического файла (см. листинг 2.8.3).

Листинг 2.8.3. Пример чтения главной загрузочной записи (Partition Table — таблица разделов)

```
.586P
;плоская модель
.MODEL FLAT, stdcall
```

[2] Напоминаю, что для того чтобы узнать причину ошибки по ее номеру, удобно воспользоваться программой errlook.exe из пакета Microsoft Visual Studio .NET.

[3] Упаси вас Бог что-либо писать вот так, не изучив скрупулезнейшим образом дисковую структуру.

```
;константы
STD_OUTPUT_HANDLE equ -11
GENERIC_READ      equ 80000000h
FILE_SHARE_WRITE  equ 2h
OPEN_EXISTING     equ 3
;прототипы внешних процедур
EXTERN  GetLastError@0:NEAR
EXTERN  wsprintfA:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  ReadFile@20:NEAR
EXTERN  CreateFileA@28:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  CloseHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  GetStdHandle@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;----------------------------------------------
;сегмент данных
_DATA SEGMENT
      H       DD 0
      NN      DD 0
;имя устройства (первый диск)
      PATHM   DB "\\.\PhysicalDrive0",0
      ALIGN   4  ;выравнивание по границе двойного слова
      BUF     DB 512 DUP(0)
      BUF1    DB 24  DUP(0)
      HANDL   DD 0
      LENS    DD 0
      ERRS    DB "Error %u ",0
      SINGL   DB "Signature %x ",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;открыть физический диск
      PUSH 0                 ;должен быть равен 0
      PUSH 0                 ;атрибут файла - не имеет значения
      PUSH OPEN_EXISTING     ;как открывать
      PUSH 0                 ;указатель на security_attr = NULL
      PUSH FILE_SHARE_WRITE  ;режим общего доступа
      PUSH GENERIC_READ      ;режим доступа - чтение
```

```
      PUSH OFFSET PATHM       ;имя устройства
      CALL CreateFileA@28
      CMP  EAX,-1
      JNZ  NO_ERR
ER:
;получить номер ошибки
      CALL GetLastError@0
      ;MOVZX EAX,AX
      PUSH EAX
      PUSH OFFSET ERRS
      PUSH OFFSET BUF1
      CALL wsprintfA
;вывести номер ошибки
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
      JMP  EXI
NO_ERR:
      MOV  H,EAX
;чтение из Partition Table
      PUSH 0
      PUSH OFFSET NN
      PUSH 512
      PUSH OFFSET BUF
      PUSH H
      CALL ReadFile@20
      CMP  EAX,0
      JZ   ER
;вывод сигнатуры Partition Table
;должно быть aa55
      PUSH DWORD PTR BUF+510
      PUSH OFFSET SINGL
      PUSH OFFSET BUF1
      CALL wsprintfA
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
;закрыть устройство
      PUSH H
      CALL CloseHandle@4
;выход
EXI:
      PUSH 0
      CALL ExitProcess@4
;вывести строку (в конце перевод строки)
;EAX - на начало строки
```

```
;EDI - с переводом строки или без
WRITE   PROC
;получить длину параметра
      PUSH  EAX
      PUSH  EAX
      CALL  lstrlenA@4
      MOV   ESI,EAX
      POP   EBX
      CMP   EDI,1
      JNE   NO_ENT
;в конце - перевод строки
      MOV  BYTE PTR [EBX+ESI],13
      MOV  BYTE PTR [EBX+ESI+1],10
      MOV  BYTE PTR [EBX+ESI+2],0
      ADD  EAX,2
NO_ENT:
;вывод строки
      PUSH 0
      PUSH OFFSET LENS
      PUSH EAX
      PUSH EBX
      PUSH HANDL
      CALL WriteConsoleA@20
      RET
WRITE  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.8.3:

```
ML /c /coff prog.ASM
LINK /SUBSYSTEM:CONSOLE prog.OBJ
```

А теперь комментарий к программе из листинга 2.8.3.

- Обратите внимание, что начало буфера, куда будет считываться сектор, должен быть выровнен по четырехбайтной границе. Для того чтобы гарантировать это, мы используем директиву `ALIGN`. Кстати, в таком выравнивании для чтения из простого файла нет необходимости, разве только вы отмените кэширование данных (см. описание функции `CreateFile` в начале главы).
- В данной программе мы впервые применили функцию `GetLastError`. В случае возникновения ошибки при открытии устройства или чтении из него программа выведет на консоль код ошибки. В документации можно будет найти, что означает эта ошибка. Можете попробовать поизменять

параметры функций `CreateFile` и `ReadFile` и посмотреть, какие коды ошибок будут выдаваться — это довольно поучительное занятие.

- Наша программа читает первый сектор диска и выводит его сигнатуру. Сигнатура всегда должна быть равна `aa55h`.

Обзор некоторых других функций API, используемых для управления файлами

Здесь мы кратко рассмотрим некоторые функции API, имеющие отношение к управлению файлами и еще подробно не описанные.

- Функция `CreateDirectory`. С помощью данной функции можно создать каталог. Функция имеет два параметра. Первый параметр указывает на строку, содержащую имя создаваемого каталога, второй — на атрибут безопасности (см. подробнее далее). Обычно второй параметр полагают равным нулю.
- Функция `RemoveDirectory`. Функция удаляет каталог. Единственным параметром функции является адрес строки, содержащей имя удаляемого каталога. Для удаления необходимо, чтобы каталог был пуст.
- Функция `SetCurrentDirectory`. Функция устанавливает текущий каталог. Ее параметром является адрес строки, содержащей имя каталога. Следует иметь в виду, что у каждого раздела может быть свой текущий каталог.
- Функция `GetCurrentDirectory`. С помощью этой функции можно получить текущий каталог. Функция имеет два параметра. Второй параметр — это адрес строки (буфера), где будет помещено имя текущего каталога. Первый параметр должен быть равен длине буфера. Предполагается, что строка будет завершаться символом с кодом 0. Следовательно, длина буфера должна рассчитываться с учетом и последнего нулевого символа.
- Функция `DeleteFile`. Функция удаляет заданный файл. Единственный параметр функции должен содержать адрес строки, определяющей имя файла. Функция не будет удалять открытый файл или файл, имеющий атрибут, который запрещает удаление.
- Функция `CopyFile`. Функция копирования файлов. Первый параметр является адресом строки, содержащей имя копируемого файла. Второй параметр указывает на строку, содержащую новое имя файла. Третий параметр определяет поведение функции, если файл с таким именем и в заданном каталоге уже существует. Если параметр равен 0, то существующий файл (если он есть) будет заменен копируемым. Если параметр имеет ненулевое значение, то функция не будет перезаписывать уже существующий файл.

- Функция `MoveFile`. Функция перемещает файл или каталог с подкаталогами. Первым аргументом функция является имя файла или каталога, вторым аргументом — новое имя файла или каталога.
- Функция `MoveFileEx`. Функция перемещает файл или каталог с учетом значения третьего аргумента. Первые два аргумента — соответственно, имя старого файла или каталога и имя нового файла или каталога. Третий аргумент может принимать следующие значения:
  - `MOVEFILE_REPLACE_EXISTING = 1` — заменять существующий файл;
  - `MOVEFILE_WRITE_THROUGH = 8` — гарантирует, что функция не возвратит управление, пока файл не будет переписан из промежуточного буфера на диск;
  - `MOVEFILE_COPY_ALLOWED = 2` — указывает, что когда новый файл будет находиться на другом томе, перемещение осуществляется выполнением двух функций `CopyFile` и `DeleteFile`;
  - `MOVEFILE_DELAY_UNTIL_REBOOT = 4` — задерживает перемещение файла до перезапуска системы. Разрешен только для администратора.
- Функция `SetFilePointer`. Установка указателя файла на произвольную позицию. Первым параметром является дескриптор открытого файла. Вторым параметром является младшая часть (32 бита) значения перемещения указателя. Третий параметр — старшая часть значения перемещения указателя. Четвертый параметр определяет режим перемещения:
  - `FILE_BEGIN = 0` — позиция отсчитывается от начала файла;
  - `FILE_CURRENT = 1` — положение вычисляется, исходя из текущей позиции;
  - `FILE_END = 2` — положение указателя определяется по отношению к концу файла.

  Замечу, что перемещение может осуществляться как по направлению к концу, так и по направлению к началу. Таким образом, значение перемещения интерпретируется как знаковое 32-битное число.
- Функция `SetEndOfFile`. Функция переносит конец файла в текущую позицию. Единственным аргументом функции является дескриптор файла. Функция может как усекать файл, так и удлинять его.

Асинхронный ввод/вывод

Ввод/вывод — это взаимодействие процесса с некоторым устройством, например файлом, хранящимся на диске. Скорость этой операции зависит не столько от производительности центрального процессора, сколько от произ-

водительности внешнего устройства и канала передачи. Обычный способ взаимодействия с внешним устройством предполагает, что процесс будет на каждом этапе взаимодействия ожидать выполнения операции записи или чтения. Такой способ называют *синхронным вводом/выводом*. Синхронным он называется потому, что процесс на каждом этапе синхронизирует свои действия с состоянием внешнего устройства. Для того чтобы увеличить производительность системы при операциях ввода/вывода, используют *асинхронный ввод/вывод*[4]. Для того чтобы уведомить систему о намерении осуществить асинхронный ввод/вывод, следует открыть устройство с помощью функции `CreateFile`, указав в шестом параметре флаг `FILE_FLAG_OVERLAPPED`. При выполнении функций `RedaFile` и `WriteFile` драйвер устройства ставит запрос ввода/вывода в очередь. При этом функции тут же возвращают управление вызвавшему их процессу. Процесс, таким образом, может выполнять другие действия и в принципе не заботиться о том, переданы данные или нет. Во всей этой схеме самое важное — это то, как драйвер даст знать процессу, что передача данных уже завершена и с каким результатом.

При асинхронном вводе/выводе несколько меняется роль параметров функций `ReadFile` и `WriteFile`. В частности, поскольку обе функции сразу возвращают управление, то четвертый параметр перестает играть какую-либо роль, и его обычно полагают равным 0. И наоборот, главную роль начинает играть пятый параметр. Он должен указывать на структуру `OVERLAPPED`, к разбору которой мы сейчас и приступаем.

```
OVERLAPPED STRUC
      INTERN DD ?
      NBYTE  DD ?
      OFFSL  DD ?
      OFFSH  DD ?
      HEVENT DD ?
OVERLAPPED ENDS
```

Замечу, что последним трем полям следует задать значения до выполнения операции. Дадим описание полям этой структуры.

- `INTERN` — это поле будет содержать статус выполнения операции. Пока операция ввода/вывода будет выполняться, значение этого поля будет равно `STATUS_PENDING = 103h`.
- `NBYTE` — по завершению операции это поле будет содержать количество реально переданных байтов.

---

[4] Для понимания данного раздела понадобится материал из *главы 3.2*, в ней также будет приведен пример параллельной асинхронной обработки.

- `OFFSL` и `OFFSH` — соответственно младшая и старшие части положения указателя в файле. Для нефайловых устройств их следует устанавливать равными 0.
- `HEVENT` — принимает значение в зависимости от метода асинхронного ввода/вывода *(см. далее)*.

Следует иметь в виду, что функции `ReadFile` и `WriteFile` при асинхронном вводе/выводе возвращают нулевое значение. Однако иногда система может посчитать, что сразу способна выполнить операцию чтения-записи, и тогда эти функции возвратят значение, большее нуля. Возвращение этими функциями нулевого значения еще не означает, что асинхронный ввод/вывод начался успешно. Возможно, произошла какая-то ошибка. Поэтому следует вызвать функцию `GetLastError`. Если она возвратит значение `ERROR_IO_PENDING = 997`, то асинхронный процесс действительно начался успешно. Любое другое значение будет означать, что произошла какая-то ошибка.

Отмена асинхронного ввода/вывода возникает, если:

- выполняется функция `CancelIO`. Единственным параметром функции является дескриптор открытого устройства (файла). Имейте в виду, что отменяются все асинхронные запросы ввода/вывода, сделанные данным потоком;
- при закрытии устройства (файла) отменяются все запросы на асинхронный ввод/вывод для данного файла;
- автоматически отменяются все запросы, выданные данным потоком, при закрытии этого потока;
- существует четыре механизма уведомления процесса о завершении процесса ввода/вывода:
  - сигнализация объекта ядра[5], управляющего устройством;
  - сигнализация объекта ядра, управляющего событиями;
  - оповестительный ввод/вывод;
  - использование порта завершения ввода/вывода.

Остановимся на указанных подходах несколько подробнее.

- *Сигнализация объекта ядра, управляющего устройством.* В данном подходе используется API-функция `WaitForSingleObject` *(см. главу 3.2)*. Функция возвращает управление либо по истечению заданного интервала времени, либо когда некоторый объект ядра, а к таковым относится,

[5] Ядро — часть операционной системы, работающей в привилегированном режиме. Более подробно о ядре можно узнать в *главе 4.6 (см. разд. "Драйверы режима ядра и устройства")*.

например, открытый файл, перейдет в некоторое состояние, называемое *сигнальным*. Первым аргументом функции является дескриптор открытого устройства (файла), вторым аргументом — интервал ожидания. Если в качестве второго аргумента взять константу `INFINITE = -1`, то интервал ожидания события будет бесконечным. Функции `ReadFile` и `WriteFile` устанавливают объект в несигнальное состояние. По окончанию асинхронного ввода/вывода драйвер устройства переводит объект в сигнальное состояние, и таким образом функция `WaitForSingleObject` возвращает управление. Вы можете спросить меня: "В чем же здесь выигрыш, ведь процесс попрежнему ожидает результата?" Обычно для использования такого метода создают отдельный поток, который и ждет завершения асинхронной операции. Поток же, открывший устройство, может заниматься другими делами. По завершению процесса следует проверить структуру `OVERLAPPED` на предмет того, не было ли ошибки и сколько байтов было передано. Перед открытием устройства эта структура должна быть проинициализирована. В частности, поле `HEVENT` должно быть инициализировано нулем.

- *Сигнализация объекта ядра, управляющего событиями.* Описанный выше способ, по сути, позволяет ожидать в данный момент только результат одной операции ввода/вывода. Более гибким будет подход, когда с каждой операцией ввода/вывода (функцией `WriteFile` или `ReadFile`) будет связано некоторое событие. Событие может быть создано с помощью `CreateEvent` *(см. главу 3.2)*, и дескриптор его нужно присвоить полю `HEVENT` структуры `OVERLAPPED`. Для ожидания события используется функция `WaitForMultipleObjects`.
- *Оповестительный ввод/вывод.* Для применения этого метода вместо функций `ReadFile` и `WriteFile` следует использовать функции `ReadFileEx` и `WriteFileEx`. Эти функции имеют дополнительный шестой параметр, представляющий адрес процедуры, которая будет вызвана, когда закончится асинхронный ввод/вывод. Эта процедура должна быть предварительно создана в вашей программе. Обычно такую процедуру называют *процедурой обратного вызова*. Она имеет три параметра. Первый параметр будет содержать код завершения. Если значение равно нулю, то асинхронный ввод/вывод закончился успешно. Если параметр равен `ERROR_HANDLE_EOF = 38`, то была произведена попытка чтения за границей файла. При успешном завершении операции следующий параметр должен содержать количество прочитанных или записанных байтов. Наконец, третий параметр — это адрес структуры `OVERLAPPED`, о которой мы уже говорили. Надо иметь в виду, что драйвер устройства по завершению очередной операции асинхронного ввода/вывода ставит соответствующую процедуру в очередь. Для того чтобы эти процедуры были вызваны, поток должен перейти

в состояние ожидания. Для этого нужно вызвать функции `SleepEx`, `WaitFoeSingleObjectEx` и др. Остановимся подробнее на функции `SleepEx`[6]. Первым параметром этой функции является количество миллисекунд ожидания. Значение параметра `INFINITE` означает бесконечно долгое ожидание. Второй параметр может принимать два значения: 0 или 1. Если значение этого параметра равно 0, то выход из данной функции осуществляется только по завершению интервала, указанного в первом параметре. Если параметр равен 1, выход из данной функции может произойти еще по двум причинам:

- при вызове функции обратного вызова;
- при установке этой функции в очередь.

❒ *Порты завершения ввода/вывода.* Рассматриваемый здесь асинхронный ввод/вывод в действительности не так часто используется для файловой обработки. Куда более актуальным является асинхронное взаимодействие между программой-сервером и несколькими клиентскими приложениями. Реальный механизм взаимодействия не так важен. Это могут быть именованные каналы, почтовые ящики или сетевое взаимодействие посредством сокетов. Здесь возможны два подхода:

- единственный поток ожидает запроса от клиента. При появлении запроса поток обрабатывает его. Такой подход хорош при редких запросах. Если к серверу обращается несколько клиентских программ одновременно, то возникает очередь, и некоторые клиенты могут достаточно долго ожидать, когда сервер начнет работать с ними. Этот подход называют *последовательной обработкой*;
- поток сервера ожидает прихода запроса от клиента, когда приходит запрос, то создается новый поток, на который возлагается обязанность взаимодействия с данным клиентом. Первый же поток продолжает ожидать новых клиентских запросов и при появлении таковых создает для них новые потоки. Поток, заканчивающий работу с клиентом, прекращает свое существование. Такой подход можно назвать *параллельной обработкой*. Он гораздо более эффективен, чем последовательный подход. Исследования, однако, показали, что при одновременном выполнении большого количества таких потоков основное время тратится на осуществление переключений контекстов потоков.

Метод, использующий порт завершения ввода/вывода, основывается на предположении, что имеется только ограниченное число одновременно работающих с клиентами потоков. Другими словами, имеется золотая се-

---

[6] В дальнейшем мы познакомимся с более простой функцией `Sleep`.

редина между параллельным и последовательным подходом. В этом методе имеется и еще один интересный подход: поскольку на создание отдельных потоков также требуется время, то создается целый пул потоков. Потоки, не участвующие в обработке клиентских запросов, находятся в состоянии ожидания.

Мы заканчиваем рассмотрение асинхронного ввода/вывода и возвратимся к нему, когда будем рассматривать многозадачное программирование *(см. главу 3.2)*.

Запись в файл дополнительной информации

Ранее мы уже говорили, что в файловой системе NTFS с именем файла можно связать несколько информационных потоков. Обычно мы используем только один поток "по умолчанию". Добавляя туда другие потоки, можно хранить в одном файле и некоторую скрытую информацию, необходимую для работы с ним. Сейчас мы рассмотрим простой пример (см. листинг 2.8.4), в котором в создаваемом файле кроме потока по умолчанию формируется еще один поток с дополнительной информацией.

Листинг 2.8.4. Создание файла с двумя потоками информации

```
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
GENERIC_READ      equ 80000000h
GENERIC_WRITE     equ 40000000h
GEN = GENERIC_READ or GENERIC_WRITE
SHARE = 0
OPEN_EXISTING     equ 3
CREATE_ALWAYS     equ 2
;прототипы внешних процедур
EXTERN  ExitProcess@4:NEAR
EXTERN  CreateFileA@28:NEAR
EXTERN  CloseHandle@4:NEAR
EXTERN  WriteFile@20:NEAR
;-----------------------------------------------
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
```

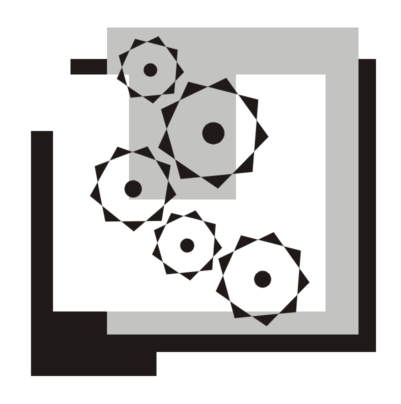
```
;сегмент данных
_DATA SEGMENT
      HAND1 DWORD ?
      HAND2 DWORD ?
      BUF1  DB "Поток 1",0
      BUF2  DB "Поток 2",0
      NAME1 DB "streams.txt",0
      NAME2 DB "streams.txt:stream1:$DATA",0
      NUMB  DD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;создать файл
      PUSH 0
      PUSH 0
      PUSH CREATE_ALWAYS
      PUSH 0
      PUSH 0
      PUSH GEN
      PUSH OFFSET NAME1
      CALL CreateFileA@28
      MOV HAND1,EAX
;запись
      PUSH 0
      PUSH OFFSET NUMB
      PUSH 7
      PUSH OFFSET BUF1
      PUSH HAND1
      CALL WriteFile@20
;закрыть файл
      PUSH HAND1
      CALL CloseHandle@4
;создать еще один поток в файле
      PUSH 0
      PUSH 0
      PUSH CREATE_ALWAYS
      PUSH 0
      PUSH 0
      PUSH GEN
      PUSH OFFSET NAME2
      CALL CreateFileA@28
      MOV HAND2,EAX
;запись
      PUSH 0
      PUSH OFFSET NUMB
```

```
        PUSH 7
        PUSH OFFSET BUF2
        PUSH HAND2
        CALL WriteFile@20
;закрыть файл
        PUSH HAND2
        CALL CloseHandle@4
;конец работы программы
        PUSH 0
        CALL ExitProcess@4
_TEXT ENDS
END START
```

Трансляция программы из листинга 2.8.4:

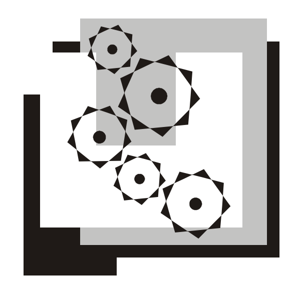
```
ML /c /coff prog.ASM
LINK /SUBSYSTEM:CONSOLE prog.OBJ
```

В листинге 2.8.4 представлена чрезвычайно простая программа, показывающая, как создавать потоки для уже существующего файла. Еще один поток для файла создается ровно по той же схеме, что и обычный файл. Просто имя файла при этом имеет следующую структуру `<имя_файла>:<имя_потока>:<тип_данных>`. Получить информацию из таких потоков можно, открыв его с помощью функции `CreateFile` и прочитав как обычный файл. В нашем случае получить содержимое второго потока можно, используя обычную программу wordpad.exe: `wordpad.exe streams.txt:stream1`.



Часть III

СЛОЖНЫЕ ПРИМЕРЫ ПРОГРАММИРОВАНИЯ В WINDOWS



Глава 3.1

Таймер в оконных приложениях

Таймер является одним из мощных инструментов, предоставляемых операционной системой и позволяющих решать самые разнообразные задачи. С таймером вы познакомились, когда занимались консольными приложениями. Там мы пользовались функциями `timeSetEvent` и `timeKillEvent`. Для консольных приложений это очень удобные функции. В оконных приложениях для создания таймеров чаще используют функции `SetTimer` и `KillTimer`. Именно эти функции станут предметом нашего изучения в данной главе.

Общие сведения

Особенность таймера, создаваемого функцией `SetTimer`, заключается в том, что сообщение `WM_TIMER`, которое начинает посылать система приложению после выполнения функции `SetTimer`, приходит со всеми другими сообщениями наравне, на общих основаниях. Следовательно, интервал между двумя приходами сообщения `WM_TIMER` может несколько варьироваться. Для большинства приложений, однако, это не существенно.

У сообщения таймера есть еще одна особенность. Если система посылает сообщение приложению, а предыдущее сообщение еще стоит в очереди, то система объединяет эти два сообщения. Таким образом, "вынужденный простой" не приводит к приходу в приложение подряд нескольких сообщений таймера. Другими словами, интервал между приходами двух сообщений `WM_TIMER` может только увеличиваться.

Вот те задачи, которые можно решить с помощью таймера:

- отслеживание времени — секундомер, часы и т. д. Нарушение периодичности не имеет значения, т. к. после прихода сообщения правильное время можно узнать, вызвав функцию получения системного времени и, таким образом, скорректировать часы;

- таймер — один из способов осуществления многозадачности. Можно установить сразу несколько таймеров на разные функции, в результате периодически будет исполняться то одна, то другая функция. Более подробно о многозадачности будет сказано в следующей главе;
- периодический вывод на экран обновленной информации или периодическое обновление некоторой информации;
- автосохранение — осуществляет периодическое сохранение данных, особенно полезно для редакторов;
- задание темпа изменения каких-либо объектов на экране;
- мультипликация — после прихода сообщения от таймера обновляется графическое содержимое экрана или окна, так что возникает эффект мультипликации.

Рассмотрим, как нужно работать с функцией создания таймера `SetTimer`. Вот параметры этой функции:

- 1-й параметр — дескриптор окна, с которым ассоциируется таймер. Если этот параметр сделать равным `NULL` (0), то будет проигнорирован и второй параметр;
- 2-й параметр определяет идентификатор таймера — ненулевое число;
- 3-й параметр определяет интервал посылки сообщения `WM_TIMER`. Единица измерения — миллисекунда. Если значение интервала больше, чем `USER_TIMER_MAXIMUM = 7FFFFFFFH`, то интервал полагается равным `USER_TIMER_MAXIMUM`. Если значение интервала меньше, чем `USER_TIMER_MINIMUM = 0AH`, то его значение полагается равным `USER_TIMER_MINIMUM`;
- 4-й параметр определяет адрес функции, на которую будет приходить сообщение `WM_TIMER`. Если параметр равен `NULL`, то сообщение будет приходить на функцию окна. Разумеется, в любом случае сообщение `WM_TIMER` проходит через цикл обработки сообщений и функцию `DispatchMessage`.

Если функция выполнилась успешно, то возвращаемым значением является идентификатор таймера, который, естественно, будет совпадать со вторым параметром, если первый параметр будет отличным от `NULL`. В случае неудачи функция возвратит ноль.

Из сказанного следует, что функция может быть вызвана тремя способами:

- задан дескриптор окна, а четвертый параметр задается равным нулю;
- задан дескриптор окна, а четвертый параметр определяет функцию, на которую будет приходить сообщение `WM_TIMER`;

- дескриптор окна равен `NULL`, а четвертый параметр определяет функцию, на которую будет приходить сообщение `WM_TIMER`. Идентификатор таймера в этом случае будет определяться по возвращаемому функцией значению.

Функция, на которую приходит сообщение `WM_TIMER`, имеет следующие параметры:

- 1-й параметр — дескриптор окна, с которым ассоциирован таймер;
- 2-й параметр — сообщение `WM_TIMER`;
- 3-й параметр — идентификатор таймера;
- 4-й параметр — время в миллисекундах, прошедшее с момента запуска Windows.

Как видим, и это понятно, параметры функции совпадают с параметрами функции окна.

Функция `KillTimer` удаляет созданный таймер и имеет следующие параметры:

- 1-й параметр — дескриптор окна;
- 2-й параметр — идентификатор таймера.

Простейший пример использования таймера

Первая программа, рассматриваемая в данном разделе, представляет простейший пример таймера (листинг 3.1.1). Таймер отсчитывает десять тиков и закрывает диалоговое окно, выдавая окно `MessageBox` с сообщением об окончании работы программы. Данная программа представляет собой пример организации таймера на базе самой процедуры окна. Диалоговое окно приложения из листинга 3.1.1 представлено на рис. 3.1.1.

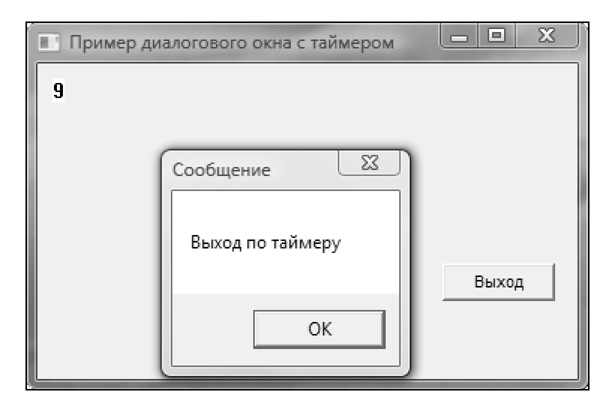


Рис. 3.1.1. Диалоговое окно с таймером

Листинг 3.1.1. Пример реализации простейшего таймера

```
//файл timer.rc
//определение констант
#define WS_SYSMENU     0x00080000L
#define WS_MINIMIZEBOX 0x00020000L
#define WS_MAXIMIZEBOX 0x00010000L
//стиль - кнопка
#define BS_PUSHBUTTON  0x00000000L
//кнопка в окне должна быть видимой
#define WS_VISIBLE     0x10000000L
//центрировать текст на кнопке
#define BS_CENTER      0x00000300L
//стиль кнопки
#define WS_CHILD       0x40000000L
//возможность фокусировать элемент
//при помощи клавиши <Tab>
#define WS_TABSTOP     0x00010000L
#define DS_3DLOOK      0x0004L
//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 120
STYLE WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX |DS_3DLOOK
CAPTION "Пример диалогового окна с таймером"
FONT 8, "Arial"
{
//кнопка, идентификатор 5
      CONTROL "Выход", 5, "button", BS_PUSHBUTTON
      | BS_CENTER | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
      180, 76, 50, 14
}

;файл timer.inc
;константы
;сообщение приходит при закрытии окна
WM_CLOSE      equ 10h
WM_INITDIALOG equ 110h
WM_COMMAND    equ 111h
WM_TIMER      equ 113h
;прототипы внешних процедур
EXTERN  ReleaseDC@8:NEAR
EXTERN  GetDC@4:NEAR
EXTERN  TextOutA@20:NEAR
EXTERN  MessageBoxA@16:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
```

```
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  SendMessageA@16:NEAR
EXTERN  SetTimer@16:NEAR
EXTERN  KillTimer@8:NEAR
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS

;файл timer.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include timer.inc
;директивы компоновщику для подключения библиотек
;для компоновщика LINK.EXE
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\gdi32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG      MSGSTRUCT <?>
      HINST    DD 0 ;дескриптор приложения
      PA       DB "DIAL1",0
      COUNT    DD 0
      TEXT     DB 0
      CAP      DB 'Сообщение',0
      MES      DB 'Выход по таймеру',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
;-------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
```

```
      PUSH  0
      PUSH  OFFSET PA
      PUSH  [HINST]
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
KOL:
;-----------------------------------
      PUSH  0
      CALL  ExitProcess@4
;---------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;-----------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
;здесь реакция на закрытие окна
L3:
;удалить таймер
      PUSH 1 ;идентификатор таймера
      PUSH DWORD PTR [EBP+08H]
      CALL KillTimer@8
;закрыть диалог
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP  FINISH
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  L5
;здесь начальная инициализация
;установить таймер
      PUSH 0       ; параметр = NULL
      PUSH 1000    ; интервал 1 секунда
      PUSH 1       ; идентификатор таймера
      PUSH DWORD PTR [EBP+08H]
```

```
      CALL SetTimer@16
      JMP  FINISH
L5:
      CMP  DWORD PTR [EBP+0CH],WM_COMMAND
      JNE  L2
;кнопка выхода?
      CMP  WORD PTR [EBP+10H],5
      JNE  FINISH
      JMP  L3
L2:
      CMP  DWORD PTR [EBP+0CH],WM_TIMER
      JNE  FINISH
;не пора ли заканчивать?
      CMP  COUNT,9
;выход без предупреждения
      JA   L3
;сообщение о выходе
      JE   L4
;пришло сообщение таймера
;подготовить текст
      MOV  EAX,COUNT
      ADD  EAX,49
      MOV  TEXT,AL
;получить контекст
      PUSH DWORD PTR [EBP+08H]
      CALL GetDC@4
;запомнить контекст
      PUSH EAX
;вывести значение счетчика
      PUSH 1
      PUSH OFFSET TEXT
      PUSH 10
      PUSH 10
      PUSH EAX
      CALL TextOutA@20
;удалить контекст
      POP  EAX
      PUSH EAX
      PUSH DWORD PTR [EBP+08H]
      CALL ReleaseDC@8
;увеличить счетчик
      INC  COUNT
      JMP  FINISH
L4:
      INC  COUNT
```

```
;сообщение о выходе по таймеру
      PUSH 0
      PUSH OFFSET CAP
      PUSH OFFSET MES
      PUSH DWORD PTR [EBP+08H] ;дескриптор окна
      CALL MessageBoxA@16
      JMP  L3
FINISH:
      POP  EDI
      POP  ESI
      POP  EBX
      POP  EBP
      MOV  EAX,0
      RET  16
WNDPROC ENDP
;-------------------------------------------
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.1.1:

```
ML /c /coff timer.asm
RC timer.rc
LINK /SUBSYSTEM:WINDOWS timer.obj timer.res
```

А теперь комментарий к программе из листинга 3.1.1.

Организация таймера здесь проста и очевидна. Таймер создается при получении процедурой окна сообщения `WM_INITDIALOG`. Единственное, что может вызвать трудность понимания, — это то, как удается оставить на экране `MessageBox` и одновременно закрыть диалоговое окно. Но здесь тоже все достаточно просто: сообщение появляется при значении `COUNT=9`, а когда приходит следующее сообщение, то `COUNT` уже больше 9, и выполняется часть кода, которая закрывает диалоговое окно.

Взаимодействие таймеров

Следующая программа несколько сложнее предыдущей. Здесь действуют два таймера. Можно считать, что запускаются одновременно две задачи[1]. Одна задача с периодичностью 0,5 секунд получает системное время и формирует строку для вывода (`STRCOPY`). Эта задача имеет собственную функцию, на которую приходит сообщение `WM_TIMER`. Вторая задача работает в рамках функции окна. Эта задача с периодичностью 1 секунда выводит время и дату

[1] Вообще говоря, три, т. к. само диалоговое окно также работает независимо.

в поле редактирования, расположенное в диалоговом окне. Таким образом, две задачи взаимодействуют друг с другом посредством глобальной переменной `STRCOPY`.

Еще один важный момент хотелось бы отметить в связи с данной программой. Поскольку в функцию таймера приходит сообщение, в котором указан идентификатор таймера, мы можем на базе одной функции реализовать любое количество таймеров. Текст программы можно видеть в листинге 3.1.2. На рис. 3.1.2 представлено окно с таймером и часами.

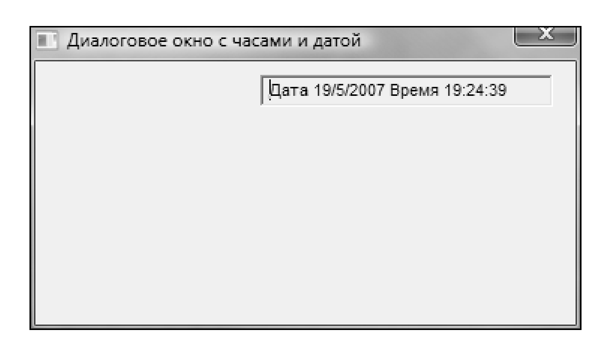


Рис. 3.1.2. Результат работы программы, представленной в листинге 3.1.2

Листинг 3.1.2. Пример взаимодействия двух таймеров

```
//файл timer2.rc
//определение констант
#define WS_SYSMENU   0x00080000L
//элементы на окне должны быть изначально видимы
#define WS_VISIBLE   0x10000000L
//бордюр вокруг элемента
#define WS_BORDER    0x00800000L
//при помощи клавиши <Tab> можно по очереди активизировать элементы
#define WS_TABSTOP   0x00010000L
//текст в окне редактирования прижат к левому краю
#define ES_LEFT      0x0000L
//стиль всех элементов в окне
#define WS_CHILD     0x40000000L
//запрещается ввод с клавиатуры
#define ES_READONLY  0x0800L

#define DS_3DLOOK    0x0004L
//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 100
STYLE WS_SYSMENU | DS_3DLOOK
```

```
CAPTION "Диалоговое окно с часами и датой"
FONT 8, "Arial"
{
      CONTROL "", 1, "edit", ES_LEFT | WS_CHILD
      | WS_VISIBLE | WS_BORDER
      | WS_TABSTOP | ES_READONLY, 100, 5, 130, 12
}

;файл timer2.inc
;константы
;сообщение приходит при закрытии окна
WM_CLOSE       equ 10h
;сообщение приходит при создании окна
WM_INITDIALOG  equ 110h
;сообщение приходит при событии с элементом в окне
WM_COMMAND     equ 111h
;сообщение от таймера
WM_TIMER       equ 113h
;сообщение посылки текста элементу
WM_SETTEXT     equ 0Ch
;прототипы внешних процедур
EXTERN  SendDlgItemMessageA@20:NEAR
EXTERN  wsprintfA:NEAR
EXTERN  GetLocalTime@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  SetTimer@16:NEAR
EXTERN  KillTimer@8:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;структура данных дата-время
DAT STRUC
      year     DW ?
      month    DW ?
      dayweek  DW ?
      day      DW ?
```

```
      hour     DW ?
      min      DW ?
      sec      DW ?
      msec     DW ?
DAT ENDS

;файл timer2.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include timer2.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\gdi32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG      MSGSTRUCT <?>
      HINST    DD 0 ;дескриптор приложения
      PA       DB "DIAL1",0
      TIM      DB "Дата %u/%u/%u Время %u:%u:%u",0
      STRCOPY  DB 50 DUP(?)
      DATA     DAT <0>
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV  [HINST], EAX
;создать диалоговое окно
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  [HINST]
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
;сообщение об ошибке
KOL:
;--------------------------------
      PUSH  0
      CALL  ExitProcess@4
;--------------------------------
```

```
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;----------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
;здесь реакция на закрытие окна
;удалить таймер 1
      PUSH 1 ;идентификатор таймера
      PUSH DWORD PTR [EBP+08H]
      CALL KillTimer@8
;удалить таймер 2
      PUSH 2 ;идентификатор таймера
      PUSH DWORD PTR [EBP+08H]
      CALL KillTimer@8
;закрыть диалог
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP  FINISH
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  L2
;здесь начальная инициализация
;установить таймер 1
      PUSH 0 ;параметр = NULL
      PUSH 1000    ; интервал 1 секунда
      PUSH 1       ; идентификатор таймера
      PUSH DWORD PTR [EBP+08H]
      CALL SetTimer@16
;установить таймер 2
      PUSH OFFSET TIMPROC ; параметр = NULL
      PUSH 500     ; интервал 0.5 секунд
      PUSH 2       ; идентификатор таймера
      PUSH DWORD PTR [EBP+08H]
      CALL SetTimer@16
      JMP FINISH
```

```
L2:
      CMP DWORD PTR [EBP+0CH],WM_TIMER
      JNE FINISH
;отправить строку в окно
      PUSH OFFSET STRCOPY
      PUSH 0
      PUSH WM_SETTEXT
      PUSH 1     ; идентификатор элемента
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
FINISH:
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      MOV EAX,0
      RET 16
WNDPROC ENDP
;-----------------------------
;процедура таймера
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM - промежуток запуска Windows
; [EBP+10H]   ;WAPARAM - идентификатор таймера
; [EBP+0CH]   ;WM_TIMER
; [EBP+8]     ;HWND
TIMPROC PROC
      PUSH EBP
      MOV EBP,ESP
;получить локальное время
      PUSH OFFSET DATA
      CALL GetLocalTime@4
;получить строку для вывода даты и времени
      MOVZX EAX,DATA.sec
      PUSH  EAX
      MOVZX EAX,DATA.min
      PUSH  EAX
      MOVZX EAX,DATA.hour
      PUSH  EAX
      MOVZX EAX,DATA.year
      PUSH  EAX
      MOVZX EAX,DATA.month
      PUSH  EAX
      MOVZX EAX,DATA.day
      PUSH  EAX
      PUSH  OFFSET TIM
      PUSH  OFFSET STRCOPY
      CALL  wsprintfA
```

```
;восстановить стек
      ADD ESP,32
      POP EBP
      RET 16
TIMPROC  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.1.2:

```
ML /c /coff timer2.asm
RC timer2.rc
LINK /SUBSYSTEM:WINDOWS timer2.obj timer2.res
```

Обращаю ваше внимание на весьма полезную функцию `GetLocalTime`, используемую нами в программе из листинга 3.1.2. Информация, полученная с помощью этой функции (см. структуру `DAT`), легко может быть использована для самых разных целей, в том числе и для вывода на экран. Аналогично, с помощью функции `SetLocalTime` вы сможете установить текущее время. Для получения времени по Гринвичу применяется функция `GetSystemTime`, которая, соответственно, с помощью `SetSystemTime` используется для установки времени в Гринвичском выражении. Аргументом во всех этих функциях является уже упомянутая выше структура (точнее, указатель на нее).

Всплывающие подсказки

В данном разделе мы рассмотрим довольно интересный вопрос о всплывающих подсказках. Всплывающие подсказки появляются, если курсор мыши оказывается над одним из элементов управления окна. В визуальных языках программирования всплывающие подсказки организуются посредством установки соответствующих свойств объектов, расположенных на объекте-контейнере. Наша с вами задача — разработать механизм, позволяющий без каких-либо дополнительных библиотек устанавливать подсказки на любые управляющие элементы, расположенные в окне. Итак, приступаем.

- Прежде всего, заметим, что всплывающая подсказка — это всего лишь окно с определенными свойствами. Вот эти свойства: `DS_3DLOOK`, `WS_POPUP`, `WS_VISIBLE`, `WS_BORDER`. В принципе можно экспериментировать — добавлять или удалять свойства. Но без одного свойства вы никак не обойдетесь — это `WS_POPUP`. Собственно, popup можно перевести как "поплавок" *(см. главу 1.3)*. Кроме того, определение всплывающего окна в файле ресурсов не должно содержать опции `CAPTION`.
- Появление подсказки не должно менять ситуацию в диалоговом окне. Это значит, что вызов подсказки должен быть немодальным при помощи

функции `CreateDialogIndirect`. Кроме того, следует предусмотреть переустановку фокуса на диалоговое окно. Для этого достаточно в нужном месте (см. листинг 3.1.3) вызвать функцию `SetFocus`.

- Итак, подсказка — это диалоговое окно, и, следовательно, оно должно иметь свою функцию. Что должна содержать эта функция? По крайней мере, обработку трех событий: `WM_INITDIALOG`, `WM_PAINT`, `WM_TIMER`. После получения сообщения `WM_INITDIALOG` следует определить размер и положение подсказки. Кроме того, если мы предполагаем, что подсказка должна спустя некоторое время исчезать (а это, как правило, необходимо, т. к. через некоторое время подсказка начнет уже мешать работе), следует установить таймер. По получении сообщения `WM_PAINT` следует вывести в окно подсказки текст. Если определять размер окна подсказки точно по строке выводимого текста, то цвет фона подсказки будет полностью определяться цветом выводимого текста. Наконец, после прихода сообщения `WM_TIMER` мы закрываем подсказку.
- С самой подсказкой более или менее ясно. Определимся теперь, как и где будет вызываться эта подсказка. Мне более импонирует такой подход: в основном диалоговом окне определяем таймер, в функции которого и будет проверяться положение курсора. В зависимости от этого положения будет вызываться или удаляться подсказка. В функции таймера необходимо предусмотреть:
  - проверку положения курсора. Если курсор оказался на данном элементе, то вызывать подсказку. При этом желательно, чтобы подсказка появлялась с некоторой задержкой. Последнее можно обеспечить введением счетчика — вызывать подсказку, если счетчик превысил некоторое значение;
  - необходимо обеспечить удаление подсказки, если курсор покидает данный элемент.

В листинге 3.1.3 представлена программа, которая демонстрирует описанный выше механизм. На рис. 3.1.3 представлено диалоговое окно с подсказками. В принципе, описанный подход не является единственным, и, разобравшись в нем, вы сможете пофантазировать и придумать свои способы создания подсказок.

Листинг 3.1.3. Пример диалогового окна с всплывающими подсказками

```
//файл HINT.RC
//определение констант
#define WS_SYSMENU      0x00080000L
//элементы на окне должны быть изначально
```

```
//видимы
#define WS_VISIBLE     0x10000000L
//бордюр вокруг элемента
#define WS_BORDER      0x00800000L
//при помощи клавиши <Tab> можно по очереди активизировать элементы
#define WS_TABSTOP     0x00010000L
//текст в окне редактирования прижат к левому краю
#define ES_LEFT        0x0000L
//стиль всех элементов на окне
#define WS_CHILD       0x40000000L
//стиль - кнопка
#define BS_PUSHBUTTON  0x00000000L
//центрировать текст на кнопке
#define BS_CENTER      0x00000300L
//тип окна - всплывающее
#define WS_POPUP       0x80000000L
//стиль - диалоговое окно Windows 95
#define DS_3DLOOK      0x0004L
//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 100
STYLE WS_SYSMENU | DS_3DLOOK
CAPTION "Окно с всплывающими подсказками"
FONT 8, "Arial"
{
//поле редактирования, идентификатор 4
 CONTROL "", 4, "edit", ES_LEFT | WS_CHILD
 | WS_VISIBLE | WS_BORDER
 | WS_TABSTOP , 100, 5, 130, 12
//кнопка, идентификатор 3
 CONTROL "Выход", 3, "button", BS_PUSHBUTTON
 | BS_CENTER |  WS_VISIBLE | WS_TABSTOP,
 180, 76, 50, 14
}

//диалоговое окно подсказки
HINTW DIALOG 0, 0, 240, 8
STYLE     WS_VISIBLE  | WS_POPUP
FONT 8, "MS Sans Serif"
{
}

;файл HINT.INC
;константы
;цвет фона окна подсказки
RED   = 255
GREEN  = 255
```

```
BLUE   = 150
RGBB   equ (RED or (GREEN shl 8)) or (BLUE shl 16)
;цвет текста окна подсказки
RED   = 20
GREEN  = 20
BLUE   = 20
RGBT   equ (RED or (GREEN shl 8)) or (BLUE shl 16)
;сообщение приходит при закрытии окна
WM_CLOSE       equ 10h
WM_INITDIALOG  equ 110h
WM_COMMAND     equ 111h
WM_TIMER       equ 113h
WM_SETTEXT     equ 0Ch
WM_COMMAND     equ 111h
WM_PAINT       equ 0Fh
WM_LBUTTONDOWN equ 201h
WM_CHAR        equ 102h
;прототипы внешних процедур
EXTERN  ShowWindow@8:NEAR
EXTERN  CreateDialogParamA@20:NEAR
EXTERN  SetActiveWindow@4:NEAR
EXTERN  lstrcpyA@8:NEAR
EXTERN  DestroyWindow@4:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  GetDlgItem@8:NEAR
EXTERN  GetCursorPos@4:NEAR
EXTERN  TextOutA@20:NEAR
EXTERN  SetBkColor@8:NEAR
EXTERN  SetTextColor@8:NEAR
EXTERN  BeginPaint@8:NEAR
EXTERN  EndPaint@8:NEAR
EXTERN  GetTextExtentPoint32A@16:NEAR
EXTERN  MoveWindow@24:NEAR
EXTERN  GetWindowRect@8:NEAR
EXTERN  ReleaseDC@8:NEAR
EXTERN  GetDC@4:NEAR
EXTERN  SendDlgItemMessageA@20:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  SetTimer@16:NEAR
EXTERN  KillTimer@8:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
```

```
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;структура размера окна
RECT  STRUC
      L  DD ?
      T  DD ?
      R  DD ?
      B  DD ?
RECT  ENDS
;структура "размер"
SIZ  STRUC
      X DD ?
      Y DD ?
SIZ  ENDS
;структура для BeginPaint
PAINTSTR STRUC
      hdc     DWORD 0
      fErase  DWORD 0
      left    DWORD 0
      top     DWORD 0
      right   DWORD 0
      bottom  DWORD 0
      fRes    DWORD 0
      fIncUp  DWORD 0
      Reserv  DB 32 dup(0)
PAINTSTR ENDS
;структура для получения позиции курсора
POINT  STRUC
      X DD ?
      Y DD ?
POINT  ENDS

;файл HINT.ASM
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include hint.inc
;директивы компоновщику для подключения библиотек
includelib C:\masm32\lib\user32.lib
includelib C:\masm32\lib\kernel32.lib
includelib C:\masm32\lib\gdi32.lib
```

```
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG     MSGSTRUCT <?>
      HINST   DD 0 ;дескриптор приложения
      PA      DB "DIAL1",0
      HIN     DB "HINTW",0
      XX      DD ?
      YY      DD ?
;---------------------------
      R1      RECT  <?>
      R2      RECT  <?>
      S       SIZ   <?>
      PS      PAINTSTR <?>
      PT      POINT  <?>
;дескрипторы окон-подсказок для первого и второго элементов
      H1      DD 0
      H2      DD 0
;строка-подсказка
      HINTS   DB 60 DUP(?)
;перечень подсказок
      HINT1   DB "Редактирование строки",0
      HINT2   DB "Кнопка выхода",0
;для временного хранения контекста устройства
      DC      DD ?
;счетчик
      P1      DD ?
;дескриптор окна
      HWND    DD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
;---------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  HINST
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
```

```
KOL:
;---------------------------------
      PUSH  0
      CALL  ExitProcess@4
;---------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
;здесь реакция на закрытие окна
;удалить таймер
L4:
      PUSH 2 ;идентификатор таймера
      PUSH DWORD PTR [EBP+08H]
      CALL KillTimer@8
;закрыть диалог
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP FINISH
L1:
      CMP DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE L2
;здесь начальная инициализация
      MOV EAX,DWORD PTR [EBP+08H]
      MOV HWND,EAX
;установить таймер
      PUSH OFFSET TIMPROC
      PUSH 500   ; интервал 0.5 секунд
      PUSH 2     ; идентификатор таймера
      PUSH DWORD PTR [EBP+08H]
      CALL SetTimer@16
      JMP  FINISH
L2:
      CMP DWORD PTR [EBP+0CH],WM_COMMAND
      JNE L3
```

```
;кнопка выхода?
      CMP WORD PTR [EBP+10H],3
      JNE L3
      JMP L4
L3:
FINISH:
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      MOV EAX,0
      RET 16
WNDPROC ENDP
;-----------------------------------------
;процедура таймера
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM - промежуток запуска Windows
; [EBP+10H]   ;WAPARAM - идентификатор таймера
; [EBP+0CH]   ;WM_TIMER
; [EBP+8]     ;HWND
TIMPROC PROC
      PUSH EBP
      MOV  EBP,ESP
;получить положение курсора
      PUSH OFFSET PT
      CALL GetCursorPos@4
;запомнить координаты
      MOV  EAX,PT.X
      MOV  XX,EAX
      MOV  EAX,PT.Y
      MOV  YY,EAX
;получить положение элементов
;окно редактирования
      PUSH 4
      PUSH DWORD PTR [EBP+08H]
      CALL GetDlgItem@8
      PUSH OFFSET R1
      PUSH EAX
      CALL GetWindowRect@8
;кнопка выхода
      PUSH 3
      PUSH DWORD PTR [EBP+08H]
      CALL GetDlgItem@8
      PUSH OFFSET R2
      PUSH EAX
      CALL GetWindowRect@8
```

```
;увеличить счетчик
      INC P1
      MOV ECX,XX
      MOV EDX,YY
;проверка условий
.IF H1==0 && P1>5
  .IF EDX<=R1.B && EDX>=R1.T && ECX>=R1.L && ECX<=R1.R
;подготовить строку
      PUSH  OFFSET HINT1
      PUSH  OFFSET HINTS
      CALL  lstrcpyA@8
;создать диалоговое окно - подсказку
      PUSH  0
      PUSH  OFFSET HINT
      PUSH  DWORD PTR [EBP+08H]
      PUSH  OFFSET HIN
      PUSH  [HINST]
      CALL  CreateDialogParamA@20
      MOV   H1,EAX
;обнулить счетчик
      MOV   P1,0
      JMP   _END
  .ENDIF
.ENDIF
.IF H1!=0
  .IF (EDX>R1.B || EDX<R1.T) || (ECX<R1.L || ECX>R1.R)
;удаление подсказки в связи с перемещением курсора
      PUSH  H1
      CALL  DestroyWindow@4
;активизировать главное окно
      PUSH  HWND
      CALL  SetActiveWindow@4
;обнулить дескриптор
      MOV   H1,0
      JMP   _END
  .ENDIF
.ENDIF
.IF H2==0 && P1>5
  .IF EDX<=R2.B && EDX>=R2.T && ECX>=R2.L && ECX<=R2.R
;подготовить строку
      PUSH OFFSET HINT2
      PUSH OFFSET HINTS
      CALL lstrcpyA@8
;создать диалоговое окно - подсказку
      PUSH  0
      PUSH  OFFSET HINT
```

```
      PUSH  DWORD PTR [EBP+08H]
      PUSH  OFFSET HIN
      PUSH  [HINST]
      CALL  CreateDialogParamA@20
      MOV   H2,EAX
;обнулить счетчик
      MOV   P1,0
      JMP  _END
  .ENDIF
.ENDIF
.IF H2!=0
  .IF (EDX>R2.B || EDX<R2.T) || (ECX<R2.L || ECX>R2.R)
;удаление подсказки в связи с перемещением курсора
      PUSH  H2
      CALL  DestroyWindow@4
;активизировать главное окно
      PUSH  HWND
      CALL  SetActiveWindow@4
;обнулить дескриптор
      MOV   H2,0
      JMP  _END
  .ENDIF
.ENDIF
;восстановить стек
_END:
      POP  EBP
      RET  16
TIMPROC ENDP
;процедура окна всплывающей подсказки
HINT PROC
      PUSH EBP
      MOV  EBP,ESP
;----------------
      CMP  DWORD PTR [EBP+0CH],WM_CHAR
      JE  _DEL
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  NO_INIT
;инициализация
;получить контекст
      PUSH DWORD PTR [EBP+08H]
      CALL GetDC@4
      MOV  DC,EAX
;получить длину строки
      PUSH OFFSET HINTS
      CALL lstrlenA@4
```

```
;получить длину и ширину строки
      PUSH OFFSET S
      PUSH EAX
      PUSH OFFSET HINTS
      PUSH DC
      CALL GetTextExtentPoint32A@16
;установить положение и размер окна-подсказки
      PUSH 0
      PUSH S.Y
      ADD  S.X,2
      PUSH S.X
      SUB  YY,20
      PUSH YY
      ADD  XX,10
      PUSH XX
      PUSH DWORD PTR [EBP+08H]
      CALL MoveWindow@24
;закрыть контекст
      PUSH DC
      PUSH DWORD PTR [EBP+08H]
      CALL ReleaseDC@8
;установить таймер
      PUSH 0
      PUSH 2000   ; интервал 2 секунды
      PUSH 3      ; идентификатор таймера
      PUSH DWORD PTR [EBP+08H]
      CALL SetTimer@16
      JMP FIN
NO_INIT:
      CMP DWORD PTR [EBP+0CH],WM_PAINT
      JNE NO_PAINT
;перерисовка окна
;получить контекст
      PUSH OFFSET PS
      PUSH DWORD PTR [EBP+08H]
      CALL BeginPaint@8
      MOV  DC,EAX
;установить цвета фона и текста подсказки
      PUSH RGBB
      PUSH EAX
      CALL SetBkColor@8
      PUSH RGBT
      PUSH DC
      CALL SetTextColor@8
;вывести текст
      PUSH OFFSET HINTS
```

```
        CALL lstrlenA@4
        PUSH EAX
        PUSH OFFSET HINTS
        PUSH 0
        PUSH 0
        PUSH DC
        CALL TextOutA@20
;закрыть контекст
        PUSH OFFSET PS
        PUSH DWORD PTR [EBP+08H]
        CALL EndPaint@8
        JMP  FIN
NO_PAINT:
        CMP  DWORD PTR [EBP+0CH],WM_TIMER
        JNE  FIN
_DEL:
;обработка события таймера
;удалить таймер и удалить диалоговое окно
;подсказка удаляется в связи с истечением 2 секунд
        PUSH  3
        PUSH  DWORD PTR [EBP+08H]
        CALL  KillTimer@8
        PUSH  DWORD PTR [EBP+08H]
        CALL  DestroyWindow@4
;активизировать главное окно
        PUSH  HWND
        CALL  SetActiveWindow@4
FIN:
        POP EBP
        RET 16
HINT ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.1.3:

```
ML /c /coff /DMASM hint.asm
RC hint.rc
LINK /SUBSYSTEM:WINDOWS hint.obj hint.res
```

Комментарий к программе, представленной в листинге 3.1.3.

Прежде всего, обращу ваше внимание, что в этой программе мы используем условные конструкции времени выполнения. Данный шаг вполне закономерен и обусловлен только необходимостью несколько сократить объем, а также упростить читаемость программы.

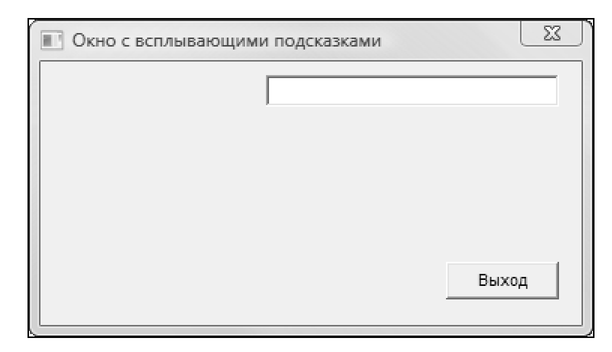
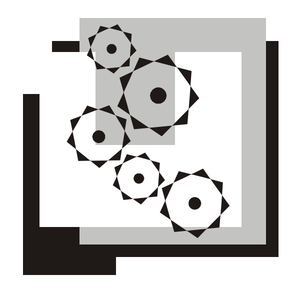


Рис. 3.1.3. Диалоговое окно с всплывающей подсказкой для поля редактирования

Как вы, наверное, уже поняли, процедура таймера проверяет каждые 0,5 секунды положение курсора. Если курсор находится на элементе (поле редактирования или кнопке) и подсказка еще не вызвана (переменные `H1` или `H2` отличны от нуля), то вызывается подсказка. При этом учитывается еще величина счетчика (переменная `P1`), чтобы подсказка появлялась с некоторой задержкой. Если при очередном вызове процедуры окажется, что курсор находится уже вне элемента, а подсказка еще на экране, то она удаляется. Данный механизм не учитывает вариант, когда курсор быстро переходит от одного элемента к другому. В этом случае вероятна ситуация, когда на экране окажутся две подсказки. Впрочем, первая подсказка должна тут же исчезнуть.

В нашей программе в диалоговом окне расположены всего два элемента: поле редактирования и кнопка. Я хотел показать, что в принципе не имеет значения, какой элемент управления есть в окне: для любого из них может быть установлена подсказка. Положение подсказки по отношению к курсору легко регулируется, и вы можете сами менять его.

Функция `GetCursorPos` получает положение курсора в абсолютных координатах относительно экрана. Здесь не возникает проблем, т. к. функция `GetWindowsRec` также получает положение элемента окна в абсолютных координатах. Предварительно нам приходится определять дескриптор элемента управления окна при помощи функции `GetDlgItem`.



Глава 3.2

Многозадачное программирование

В предыдущей главе нами были рассмотрены возможности использования таймеров в прикладной задаче. Задав один или несколько таймеров, мы принуждаем систему вызывать одну или несколько процедур в автоматическом режиме. Посредством таймеров мы тем самым можем реализовать многозадачный режим в рамках одного процесса. Более того, было показано, что такие подзадачи могут взаимодействовать друг с другом. Это и понятно, ведь все эти подзадачи разделяют одно адресное пространство, и, следовательно, информация от одной подзадачи к другой может передаваться через глобальные переменные. Это весьма интересный и сложный вопрос о том, как задачи и подзадачи могут взаимодействовать друг с другом. Мы рассмотрим его несколько позднее. Сейчас же рассмотрим многозадачность в операционной системе Windows с самого начала.

Процессы и потоки

Под *процессом* будем понимать объект ядра *(см. главу 2.8)*, создаваемый операционной системой Windows обычно при загрузке исполняемого модуля и получающий в единоличное пользование:

- виртуальную память, выделяемую для него операционной системой;
- дескрипторы открываемых им файлов;
- список загруженных им в его собственную память динамических модулей (DLL);
- созданные им потоки, исполняемые независимо друг от друга, в собственной памяти процесса.

Думаю, данное определение весьма ясно раскрывает суть понятия "процесс". Но для большинства рассматриваемых в данной главе проблем достаточно

было бы дать и более простое определение. Например, такое: всякий исполняемый модуль (exe), запущенный в операционной системе Windows, становится процессом. Следует понимать, что процесс — понятие статическое. После его появления должны быть созданы объекты, непосредственно связанные с выполнением кода. Такие объекты называются *потоками*. Английский термин, соответствующий понятию "поток" — thread. Другое значение этого слова — нить. Так что потоки иногда называют нитями.

Поток также представляет собой объект ядра. Данный объект характеризуется:

- контекстом потока, который, в частности, содержит значения регистров потока, используемых при переключении с одного потока на другой;
- стек потока, где хранятся параметры функций и значения локальных переменных.

При создании процесса операционной системой создается, по крайней мере, один поток, называемый *первичным потоком*. Он, в свою очередь, может создавать (по желанию программиста) еще потоки, которые называют *вторичными потоками* (рис. 3.2.1). Реальная многозадачность, таким образом, реализуется на уровне потоков, причем как в рамках одно процесса, так и в пределах нескольких процессов, и представляет собой выделение для каждого из выполняющихся потоков некоторого кванта времени. Выделением кванта времени для каждого из потоков занимается часть операционной системы, называемая *планировщиком*.

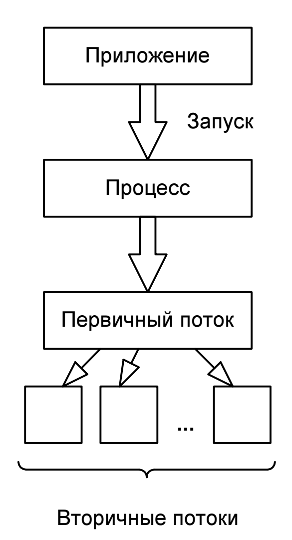


Рис. 3.2.1. Процесс и потоки

Разумеется, запущенное приложение в лице одного из потоков может создавать и целые процессы на основе того или иного приложения, которые, в свою очередь, создают собственные потоки и процессы и т. д. В результате из одного процесса может вырасти целое дерево процессов и потоков.

Параллельно выполняющиеся потоки в рамках одно процесса могут в значительной степени оптимизировать работу приложения и рационально загрузить процессор (или несколько процессоров, если таковые имеются). Запуск для выполнения отдельных подзадач целых процессов не совсем рационален, т. к. объективно процесс требует больше ресурсов, чем поток.

Теперь немного поговорим о типах многозадачности. В старой 16-битной Windows переключение между задачами происходило только тогда, когда задача отдавала управление операционной системе. Такая многозадачность называется *невытесняющей*. В определенном смысле это было даже хуже, чем в операционной системе MS-DOS. Там элементы многозадачности осуществлялись при помощи так называемых TSR-программ (см. [1]). Такие программы назывались еще *резидентными*. Они перехватывали прерывание от таймера, клавиатуры или другого устройства и имели возможность время от времени получать управление по событиям, связанным с этими устройствами.

Положение, существовавшее в старой операционной системе Windows, требовало от программиста выполнения джентльменского правила — не захватывать надолго время микропроцессора. Некоторым решением проблемы являлось использование таймеров (в чем мы уже убедились), а также использование функции `PeekMessage` вместо `GetMessage`. Функция `PeekMessage`, в отличие от `GetMessage`, возвращает управление сразу, даже если в очереди нет ни одного сообщения *(см. главу 1.2)*.

В 32-битных операционных системах Windows (Windows XP, Windows Server 2003, Windows Vista) реализована вытесняющая схема многозадачности, в которой переключением между процессами и потоками занимается операционная система. Если процесс слишком долго выполняет некоторую операцию, то курсор над окном процесса преобразуется в песочные часы. При этом другие процессы будут по-прежнему выполняться, и вы сможете переключаться на них. А вот доступ к окну данного процесса может оказаться затруднительным. Решить эту проблему можно уже упомянутым способом, заменив в цикле ожидания `GetMessage` на `PeekMessage`. Однако более правильным решением будет разбиение процесса на некоторое количество потоков.

У каждого из работающих потоков имеется характеристика, называемая *приоритетом*. В зависимости от значения этой характеристики (из промежутка 0—31) планировщик регулирует выделение квантов времени для потоков. Если у всех потоков, существующих в данный промежуток времени, приоритеты

имеют одно и то же значение, то все потоки получают один и тот же квант времени. Если приоритеты различны, то планировщик при распределении квантов времени руководствуется значениями этих приоритетов. При этом потоки с большим приоритетом получают больше машинного времени — потоки с большим приоритетом вытесняют потоки с меньшим приоритетом. Если несколько потоков с большим приоритетом работают слишком долго, то может создаться ситуация, когда потоки с меньшим приоритетом почти перестанут выполняться. По этой причине потоки с большим приоритетом не должны выполняться слишком долго. Их следует предназначать для важных действий, которые должны обязательно выполняться до некоторого момента времени[1]. Самым маленьким приоритетом 0 в системе Windows обладает служба, занимающаяся очисткой страниц памяти. Все остальные службы и программы могут иметь приоритет от 1 до 31.

При работе с потоками, однако, вам не придется работать с абсолютными значениями приоритетов. Работать приходится с относительными приоритетами, причем в два этапа. Вначале задается класс приоритета для процесса, а уже потом можно относительно класса приоритета задавать приоритет потоков. В табл. 3.2.1 представлены классы приоритетов, а в табл. 3.2.2 — относительные приоритеты.

Таблица 3.2.1. *Классы приоритетов*

Класс приоритета	**Описание**
`REALTIME_PRIORITY_CLASS = 100H`	Потоки в этом процессе должны обеспечивать критические по времени задачи. Такие потоки вытесняют даже компоненты операционных систем
`HIGH_PRIORITY_CLASS = 80H`	Как и в предыдущем случае, эти потоки предназначены для выполнения критических по времени задач
`ABOVE_NORMAL_PRIORITY_CLASS = 8000H`	Класс, промежуточный между нормальным и высоким приоритетом
`NORMAL_PRIORITY_CLASS = 20H`	Потоки в этом процессе не предъявляют особых требований к выделению им машинного времени
`BELOW_NORMAL_PRIORITY_CLASS=4000H`	Промежуточный класс между нормальным и бездействующим (idle) классами
`IDLE_PRIORITY_CLASS = 40H`	Потоки в этом процессе выполняются, если система не занята другой работой

[1] Хотя гарантированно, это можно сделать только в системах реального времени.

Таблица 3.2.2. *Относительные приоритеты*

Приоритет	Описание
THREAD_PRIORITY_ABOVE_NORMAL = 1	Поток выполняется с приоритетом, на один уровень выше обычного для данного класса
THREAD_PRIORITY_BELOW_NORMAL = -1	Поток выполняется с приоритетом, на один уровень ниже обычного для данного класса
THREAD_PRIORITY_HIGHEST = 2	Поток выполняется с приоритетом, на два уровня выше обычного для данного класса
THREAD_PRIORITY_IDLE = -15	Поток выполняется с приоритетом 16 в классе REALTIME_PRIORITY_CLASS и с приоритетом 1 в других классах
THREAD_PRIORITY_LOWEST = -2	Поток выполняется с приоритетом, на два уровня ниже обычного для данного класса
THREAD_PRIORITY_NORMAL = 0	Поток выполняется с обычным приоритетом процесса для данного класса
THREAD_PRIORITY_TIME_CRITICAL = 15	Поток выполняется с приоритетом 31 в классе REALTIME_PRIORITY_CLASS и с приоритетом 15 в других классах

Созданием потоков мы займемся в следующих разделах, а оставшаяся часть данного раздела будет посвящена созданию процессов. Ваше приложение может создавать процессы, запустив ту или иную exe-программу, которые будут работать независимо от основного приложения. Одновременно ваше приложение может при необходимости удалить запущенное им приложение из памяти. Запустить приложение (создать процесс) можно при помощи функции CreateProcess. Сейчас мы дадим описание этой функции. Ниже объясняются ее параметры.

- 1-й параметр — указывает на имя запускаемой программы. Имя может содержать полный путь к программе.
- 2-параметр — его значение зависит от того, является первый параметр NULL (0) или нет. Если первый параметр указывает на строку, то данный параметр трактуется как командная строка запуска (без имени программы). Если первый параметр равен NULL, то данный параметр рассматривается как командная строка, первый элемент которой представляет собой имя программы. Если путь к программе не указан, то функция

CreateProcess осуществляет поиск программы по определенному алгоритму:

- поиск в каталоге, где располагается запущенная программа;
- поиск в текущем каталоге. В общем случае текущий каталог может отличаться от каталога, где запускаемая программа располагается;
- поиск в системном каталоге (можно получить через GetSystemDirectory). Обычно системным каталогом является C:\Windows\System32[2];
- поиск в каталоге Windows (можно получить через GetWindowsDirectory). Обычно этим каталогом является C:\Windows;
- поиск в каталогах, перечисленных в переменной окружения PATH.

- 3-й и 4-й параметры используются для задания атрибутов доступа порождаемого процесса. Обычно их полагают равными 0.
- 5-й параметр — если этот параметр 0, то порождаемый процесс не наследует дескрипторы порождающего процесса, в противном случае порождаемый процесс наследует дескрипторы.
- 6-й параметр, в частности, задает класс приоритетов для создаваемого процесса. В качестве параметров используются флаги, представленные в табл. 3.2.1. Флаг приоритета может быть скомбинирован с флагами создания процесса (табл. 3.2.3).

Таблица 3.2.3. *Флаги создания процесса*

Флаг	Описание
CREATE_BREAKAWAY_FROM_JOB=1000000H	Создаваемый процесс не будет связан с заданием (job), если порождающий его процесс связан с некоторым заданием
CREATE_DEFAULT_ERROR_MODE=4000000H	Указывает, что порождаемый процесс не должен наследовать режимы обработки ошибок в родительском процессе
CREATE_NEW_CONSOLE=10H	Новый процесс должен создать новую консоль, вместо того чтобы наследовать консоль родительского процесса

[2] Разумеется, если каталогом, где располагается операционная система Windows, является C:\Windows.

Таблица 3.2.3 *(окончание)*

Флаг	**Описание**
`CREATE_NEW_PROCESS_GROUP=200H`	Данный флаг служит для модификации списка процессов, уведомляемых о нажатии комбинаций клавиш <Ctrl>+<C> и <Ctrl>+<Break>. Если в системе одновременно исполняются несколько GUI-процессов, то при нажатии одной из указанных комбинаций клавиш система уведомляет об этом только процессы, включенные в группу. Указав этот флаг, мы, тем самым, создаем новую группу
`CREATE_NO_WINDOW=8000000H`	Указывает, что процесс не должен содержать никаких консольных окон
`CREATE_PRESERVE_CODE_AUTHZ_LEVEL=2000000H`	Позволяет запускать дочерний процесс, на который не будут накладываться ограничения, которые должны на него накладываться по умолчанию
`CREATE_SEPARATE_WOW_VDM=800H`	Данный флаг используется при запуске 16-битных приложений, и нас интересовать не будет
`CREATE_SHARED_WOW_VDM=1000H`	Данный флаг используется при запуске 16-битных приложений, и нас интересовать не будет
`CREATE_SUSPENDED=4H`	Процесс будет создан, но главный поток будет приостановлен. Для его запуска используется функции `ResumeThread`
`CREATE_UNICODE_ENVIRONMENT=400H`	Сообщает системе, что буфер, содержащий параметры среды, должен содержать строки в кодировке Unicode
`DEBUG_ONLY_THIS_PROCESS=2H`	Дает возможность родительскому процессу проводить отладку дочернего процесса
`DEBUG_PROCESS=1H`	Дает возможность родительскому процессу проводить отладку дочернего процесса, а также всех тех, которые могут быть порождены дочерним
`DETACHED_PROCESS=8H`	Данный флаг блокирует доступ процессу, инициированному консольной программой, к созданному родительским процессом консольному окну и сообщает системе, что вывод следует перенаправить в новое окно

- 7-й параметр является указателем на буфер, содержащий параметры среды. Если параметр равен 0, то порождаемый процесс наследует параметры среды порождающего процесса. Если буфер не пуст, то он должен содержать последовательность строк вида "имя=значение", которые заканчиваются 0.
- 8-й параметр задает текущее устройство и каталог для порождаемого процесса. Если параметр равен `NULL`, порождаемый процесс наследует текущее устройство и каталог порождающего процесса.
- 9-й параметр представляет указатель на структуру, которая содержит информацию об окне создаваемого процесса. Далее будут рассмотрены поля этой структуры.
- 10-й параметр указывает на структуру, заполняемую при выполнении запуска приложения. Вот эта структура:

```
PROCINF STRUC
      hProcess  DD ? ;дескриптор созданного процесса
      hThread   DD ? ;дескриптор главного потока нового процесса
      Idproc    DD ? ;идентификатор созданного процесса
      idThr     DD ? ;идентификатор главного потока нового процесса
PROCINF ENDS
```

  Основное отличие дескриптора от идентификатора процесса заключается в том, что дескриптор уникален лишь в пределах данного процесса, идентификатор же является глобальной величиной. Посредством идентификатора может быть найдена область данных текущего процесса. У читателя, я думаю, сразу возникнет вопрос: а чем же отличается дескриптор приложения, который мы получаем при помощи функции `GetModulHandle`, от только что упомянутых величин? Так вот, дескриптор приложения и дескриптор, получаемый с помощью `GetModulHandle`, — это одно и то же. Дескриптор приложения или дескриптор модуля есть величина локальная, т. е. действующая в пределах данного процесса и, как правило, равная адресу загрузки модуля в виртуальное адресное пространство. Дескриптор модуля имеется у любого модуля, загруженного в память, в том числе и у подчиненных DLL-библиотек.

Рассмотрим теперь структуру, на которую указывает 9-й параметр функции `CreateProcess`. Вот эта структура:

```
STARTUP STRUC
      cb              DD 0
      lpReserved      DD 0
      lpDesktop       DD 0
      lpTitle         DD 0
      dwX             DD 0
      dwY             DD 0
```

```
    dwXSize         DD 0
    dwYSize         DD 0
    dwXCountChars   DD 0
    dwYCountChars   DD 0
    dwFillAttribute DD 0
    dwFlags         DD 0
    wShowWindow     DW 0
    cbReserved2     DW 0
    lpReserved2     DD 0
    hStdInput       DD 0
    hStdOutput      DD 0
    hStdError       DD 0
STARTUP ENDS
```

Итак, разберем смысл полей этой структуры:

- `cb` — размер данной структуры в байтах, заполняется обязательно;
- `lpReserved` — резерв, должно быть равно нулю;
- `lpDesktop` — имя рабочего стола (и рабочей станции). Имеет смысл только для семейства Windows NT;
- `lpTitle` — название окна для консольных приложений, создающих свое окно. Для остальных приложений должно быть равно 0;
- `dwX` — координата *X* левого верхнего угла окна;
- `dwY` — координата *Y* левого верхнего угла окна;
- `dwXSize` — размер окна по оси *x*;
- `dwYSize` — размер окна по оси *y*;
- `dwXCountChars` — размер буфера консоли по оси *x*;
- `dwYCountChars` — размер буфера консоли по оси *y*;
- `dwFillAttribute` — начальный цвет текста. Имеет значение только для консольных приложений;
- `dwFlags` — флаг значения полей (табл. 3.2.4);

***Таблица 3.2.4.** Значения флага `dwFlags`*

Макрозначение флага	**Значение константы**	**Смысл значения**
STARTF_USESHOWWINDOW	1h	Разрешить поле `dwShowWindow`
STARTF_USESIZE	2h	Разрешить поля `dwXSize` и `dwYSize`
STARTF_USEPOSITION	4h	Разрешить поля `dwX` и `dwY`

Таблица 3.2.4 (окончание)

Макрозначение флага	Значение константы	Смысл значения
STARTF_USECOUNTCHARS	8h	Разрешить поля dwXCountChars и dwYCountChars
STARTF_USEFILLATTRIBUTE	10h	Разрешить поле dwFillAttribute
STARTF_FORCEONFEEDBACK	40h	Включить возврат курсора
STARTF_FORCEOFFFEEDBACK	80h	Выключить возврат курсора
STARTF_USESTDHANDLES	100h	Разрешить поле hStdInput

- wShowWindow — определяет способ отображения окна;
- cbReserved2 — резерв, должно быть равно 0;
- hStdInput — дескриптор ввода (для консоли);
- hStdOutput — дескриптор вывода (для консоли);
- hStdError — дескриптор вывода сообщения об ошибке (для консоли).

Следующая программа (листинг 3.2.1) представляет собой простейший пример создания процесса. В качестве программы, порождающей процесс, взят редактор WINWORD.EXE. Для проверки правильности работы примера вам придется указать путь к модулю WINWORD.EXE на вашем компьютере (переменная PATH). Обратите внимание на то, что приложение появляется на экране в свернутом виде и как это достигается. Как видите, функция CreateProcess совсем не так уж страшна, как кажется на первый взгляд.

Листинг 3.2.1. Пример создания процесса

```
//файл proces.rc
//определение констант
#define WS_SYSMENU    0x00080000L
#define WS_POPUP      0x80000000L
#define DS_3DLOOK     0x0004L

//ресурс - меню
MENUP MENU
{
 POPUP "&Запуск Word"
 {
 MENUITEM "&Запустить", 1
 MENUITEM "&Удалить ", 2
 MENUITEM "Выход из &программы", 3
```

```
 }
}

//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 120
STYLE WS_POPUP | WS_SYSMENU | DS_3DLOOK
CAPTION "Пример запуска процесса"
FONT 8, "Arial"
{
}
;файл proces.inc
;константы
STARTF_USESHOWWINDOW equ 1h
SW_SHOWMINIMIZED     equ 2
;сообщение приходит при закрытии окна
WM_CLOSE             equ 10h
WM_INITDIALOG        equ 110h
WM_COMMAND           equ 111h
;прототипы внешних процедур
EXTERN TerminateProcess@8:NEAR
EXTERN CreateProcessA@40:NEAR
EXTERN DialogBoxParamA@20:NEAR
EXTERN EndDialog@8:NEAR
EXTERN MessageBoxA@16:NEAR
EXTERN ExitProcess@4:NEAR
EXTERN GetModuleHandleA@4:NEAR
EXTERN LoadMenuA@8:NEAR
EXTERN SetMenu@8:NEAR
EXTERN TranslateMessage@4:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND          DD ?
      MSMESSAGE       DD ?
      MSWPARAM        DD ?
      MSLPARAM        DD ?
      MSTIME          DD ?
      MSPT            DD ?
MSGSTRUCT ENDS
;структура для CreateProcess
STARTUP STRUC
      cb               DD 0
      lpReserved       DD 0
      lpDesktop        DD 0
      lpTitle          DD 0
      dwX              DD 0
```

```
      dwY             DD 0
      dwXSize         DD 0
      dwYSize         DD 0
      dwXCountChars   DD 0
      dwYCountChars   DD 0
      dwFillAttribute DD 0
      dwFlags         DD 0
      wShowWindow     DW 0
      cbReserved2     DW 0
      lpReserved2     DD 0
      hStdInput       DD 0
      hStdOutput      DD 0
      hStdError       DD 0
STARTUP ENDS
;структура - информация о процессе
PROCINF STRUC
      hProcess   DD ?
      hThread    DD ?
      Idproc     DD ?
      idThr      DD ?
PROCINF ENDS
;файл proces.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include proces.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;сегмент данных
_DATA SEGMENT
      NEWHWND  DD 0
      MSG      MSGSTRUCT <?>
      STRUP    STARTUP   <?>
      INF      PROCINF   <?>
      HINST    DD 0 ;дескриптор приложения
      PA       DB "DIAL1",0
      PMENU    DB "MENUP",0
      PATH     DB "C:\Program Files\Microsoft Office\Office11\WINWORD.EXE",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
```

```
      MOV  [HINST], EAX
;создать модальный диалог
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  [HINST]
      CALL  DialogBoxParamA@20
      PUSH  0
      CALL  ExitProcess@4
;--------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H] ;LPARAM
; [EBP+10H]  ;WAPARAM
; [EBP+0CH]  ;MES
; [EBP+8]    ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;сообщение при закрытии окна
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
;закрыть диалоговое окно
      JMP  L5
L1:
;сообщение при инициализации окна
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  L3
;загрузить меню
      PUSH OFFSET PMENU
      PUSH HINST
      CALL LoadMenuA@8
;установить меню
      PUSH EAX
      PUSH DWORD PTR [EBP+08H]
      CALL SetMenu@8
      JMP  FINISH
;проверяем, не случилось ли чего с пунктами
;меню в диалоговом окне
L3:
      CMP DWORD PTR [EBP+0CH],WM_COMMAND
      JNE FINISH
```

```
      CMP WORD PTR [EBP+10H],3
      JE  L5
      CMP WORD PTR [EBP+10H],2
      JE  L7
      CMP WORD PTR [EBP+10H],1
      JE  L6
      JMP FINISH
;закрыть диалоговое окно
L5:
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP  FINISH
;запустить программу Word
L6:
;заполняем структуру для запуска
;окно должно появляться в свернутом виде
      MOV STRUP.cb,68
      MOV STRUP.lpReserved,0
      MOV STRUP.lpDesktop,0
      MOV STRUP.lpTitle,0
      MOV STRUP.dwFlags,STARTF_USESHOWWINDOW
      MOV STRUP.cbReserved2,0
      MOV STRUP.lpReserved2,0
      MOV STRUP.wShowWindow,SW_SHOWMINIMIZED
;запуск приложения Word
      PUSH OFFSET INF
      PUSH OFFSET STRUP
      PUSH 0
      PUSH 0
      PUSH 0
      PUSH 0
      PUSH 0
      PUSH 0
      PUSH OFFSET PATH
      PUSH 0
      CALL CreateProcessA@40
      JMP  FINISH
;удалить из памяти процесс
L7:
      PUSH 0 ;код выхода
      PUSH INF.hProcess
      CALL TerminateProcess@8
FINISH:
      MOV EAX,0
      POP EDI
```

```
      POP ESI
      POP EBX
      POP EBP
      RET 16
WNDPROC  ENDP
_TEXT ENDS
END START
```

Трансляция программы PROCES.ASM из листинга 3.2.1:

```
ML /c /coff proces.asm
RC proces.rc
LINK /SUBSYSTEM:WINDOWS proces.obj proces.res
```

Думаю, что другой комментарий для программы в листинге 3.2.1 не требуется. Окно, выводимое программой, и раскрытое меню представлены на рис. 3.2.2.

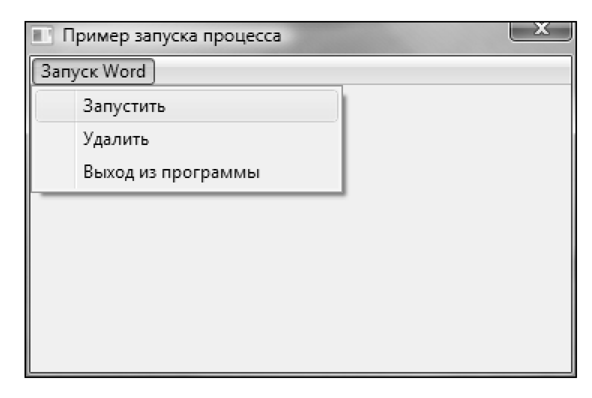


Рис. 3.2.2. Окно программы, запускающей и удаляющей из памяти программу WINWORD.EXE

Следует особо остановиться на том, как запущенный процесс может быть выгружен из памяти. Конечно, если вы запускаете уже написанный кем-то модуль, то автор (скорее всего) предусмотрел естественный выход из него. В этом случае вся ответственность по завершению приложения ложится на него. Если же вы используете функцию `TerminateProcess`, например так, как это было показано в примере из листинга 3.2.1, то здесь все может закончиться не очень хорошо. Конечно, система высвобождает все взятые процессом ресурсы, но она не может выполнять чужую работу. Например, если программа работает с файлом, то часть данных может потеряться. При использовании функции `TerminateProcess` приложению не посылаются какие-либо уведомления. Что касается функции `ExitProcess`, которую мы широко используем в данной книге, то при написании программы на ассемблере это

вполне корректный способ окончания работы процесса. Но остерегайтесь использовать эту функцию, когда пишете программу на языке высокого уровня. В вашем приложении могут быть объекты, и они будут некорректно удалены из памяти (без выполнения деструкторов).

Потоки

Теперь пришла пора вплотную заняться потоками. Вначале я намерен решить задачу из предыдущей главы (см. листинг 3.2.2) при помощи потока.

Поток может быть создан посредством функции `CreateThread`. Рассмотрим параметры этой функции:

- 1-й параметр — указатель на структуру атрибутов доступа. Обычно полагается равным `NULL`;
- 2-й параметр — размер стека потока. Если параметр равен нулю, то берется размер стека по умолчанию, равный размеру стека родительского потока;
- 3-й параметр — указатель на потоковую функцию, с вызова которой начинается исполнение потока;
- 4-й параметр — параметр для потоковой функции. Данный параметр передается потоковой функции и может быть использован для передачи некоторого инициализирующего значения;
- 5-й параметр — флаг, определяющий состояние потока. Если флаг равен 0, то выполнение потока начинается немедленно. Если значение флага потока равно `CREATE_SUSPENDED=04H`, то поток находится в состоянии ожидания и запускается по выполнению функции `ResumeThread`;
- 6-й параметр — указатель на переменную, куда будет помещен глобальный идентификатор потока. Если данный параметр задать равным 0, то тем самым вы просто отказываетесь от того, чтобы функция возвратила вам идентификатор созданного потока.

При удачном выполнении функции `CreateThread` в системе будет создан объект — поток, а сама функция возвращает дескриптор потока. Таким образом, создавая поток, мы можем получить сразу и дескриптор, и глобальный идентификатор потока. Как и в случае с процессом, идентификатор потока глобален, а дескриптор локален.

Как уже было сказано, выполнение потока начинается с потоковой функции. Окончание работы этой функции приводит к естественному окончанию работы потока (выход из функции по команде `RET`). Поток также может закончить свою работу, выполнив функцию `ExitThread` с указанием кода выхода. Наконец, порождающий поток может закончить работу порожденного потока при

помощи функции `TerminateThread`. В нашем примере в листинге 3.2.1 запускаемый процесс не может сам закончить свою работу и прекращает ее вместе с приложением по команде `TerminateThread`. Надо сказать, что такое завершение, вообще говоря, является аварийным и не рекомендуется к обычному употреблению. Связано это с тем, что при таком завершении не выполняются никакие действия по освобождению занятых ресурсов (блоки памяти, открытые файлы и т. п.). Поэтому стройте свои приложения так, чтобы поток завершался по выходу из потоковой процедуры. Таким образом, в некотором смысле приведенный далее пример является демонстрацией того, чего не рекомендуется делать.

Вообще, идеальной кажется ситуация, когда функция окна берет на себя только реакцию на события, происходящие с элементами, а всю трудоемкую работу (сложные вычисления, файловая обработка) должны взять на себя потоки. Кстати, поток может создавать новые потоки, так что в результате может возникнуть целое дерево.

Как я уже сказал, далее представлена программа (листинг 3.2.2), использующая поток для вычисления и вывода в окно редактирования текущей даты и времени. Замечу в этой связи, что если бы такая обработка была реализована в оконной функции, вы бы сразу почувствовали разницу — окно почти бы перестало реагировать на внешнее воздействие.

Листинг 3.2.2. Пример создания потока

```
//файл thread.rc
//определение констант
#define WS_SYSMENU     0x00080000L
//элементы в окне должны быть изначально видимы
#define WS_VISIBLE     0x10000000L
//бордюр вокруг элемента
#define WS_BORDER      0x00800000L
//при помощи клавиши <Tab> можно по очереди активизировать элементы
#define WS_TABSTOP     0x00010000L
//текст в окне редактирования прижат к левому краю
#define ES_LEFT        0x0000L
//стиль всех элементов в окне
#define WS_CHILD       0x40000000L
//запрещается ввод с клавиатуры
#define ES_READONLY    0x0800L
// стиль - "кнопка
#define BS_PUSHBUTTON  0x00000000L
//центрировать текст на кнопке
#define BS_CENTER      0x00000300L
```

```
#define DS_3DLOOK      0x0004L
//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 100
STYLE WS_SYSMENU | DS_3DLOOK
CAPTION "Пример использования потока"
FONT 8, "Arial"
{
//окно редактирования, идентификатор 1
 CONTROL "", 1, "edit", ES_LEFT | WS_CHILD
 | WS_VISIBLE | WS_BORDER
 | WS_TABSTOP | ES_READONLY, 100, 5, 130, 12
//кнопка, идентификатор 2
 CONTROL "Выход", 2, "button", BS_PUSHBUTTON
 | BS_CENTER | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
 180, 76, 50, 14
}

;файл thread.inc
;константы
;сообщение приходит при закрытии окна
WM_CLOSE        equ 10h
;сообщение приходит при создании окна
WM_INITDIALOG   equ 110h
;сообщение приходит при событии с элементом
;на окне
WM_COMMAND      equ 111h
;сообщение посылки текста элементу
WM_SETTEXT      equ 0Ch
;прототипы внешних процедур
EXTERN  SendMessageA@16:NEAR
EXTERN  GetDlgItem@8:NEAR
EXTERN  Sleep@4:NEAR
EXTERN  TerminateThread@8:NEAR
EXTERN  CreateThread@24:NEAR
EXTERN  wsprintfA:NEAR
EXTERN  GetLocalTime@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
```

```
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;структура данных дата-время
DAT STRUC
      year    DW ?
      month   DW ?
      dayweek DW ?
      day     DW ?
      hour    DW ?
      min     DW ?
      sec     DW ?
      msec    DW ?
DAT ENDS
;файл timer2.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include thread.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\gdi32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG     MSGSTRUCT <?>
      HINST   DD 0   ; дескриптор приложения
      PA      DB "DIAL1",0
      TIM     DB "Дата %u/%u/%u   Время %u:%u:%u",0
      STRCOPY DB 50 DUP(?)
      DATA    DAT <0>
      HTHR    DD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
;создать диалоговое окно
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
```

```
      PUSH  OFFSET PA
      PUSH  [HINST]
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
;сообщение об ошибке
KOL:
;--------------------------------
      PUSH  0
      CALL  ExitProcess@4
;--------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;----------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
L3:
;здесь реакция на закрытие окна
;удалить поток
      PUSH 0
      PUSH HTHR
      CALL TerminateThread@8
;закрыть диалог
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP  FINISH
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  L2
;здесь начальная инициализация
;получить дескриптор окна редактирования
      PUSH 1
      PUSH DWORD PTR [EBP+08H]
      CALL GetDlgItem@8
```

```
;создать поток
      PUSH OFFSET HTHR     ; сюда дескриптор потока
      PUSH 0
      PUSH EAX             ; параметр
      PUSH OFFSET GETTIME  ; адрес процедуры
      PUSH 0
      PUSH 0
      CALL CreateThread@24
      JMP  FINISH
L2:
      CMP DWORD PTR [EBP+0CH],WM_COMMAND
      JNE FINISH
;кнопка выхода?
      CMP WORD PTR [EBP+10H],2
      JE  L3
FINISH:
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      MOV EAX,0
      RET 16
WNDPROC  ENDP
;потоковая функция
; [EBP+8]  параметр=дескриптор окна редактирования
GETTIME PROC
      PUSH EBP
      MOV  EBP,ESP
LO:
;задержка в 1 секунду
      PUSH 1000
      CALL Sleep@4
;получить локальное время
      PUSH OFFSET DATA
      CALL GetLocalTime@4
;получить строку для вывода даты и времени
      MOVZX EAX,DATA.sec
      PUSH  EAX
      MOVZX EAX,DATA.min
      PUSH  EAX
      MOVZX EAX,DATA.hour
      PUSH  EAX
      MOVZX EAX,DATA.year
      PUSH  EAX
      MOVZX EAX,DATA.month
      PUSH  EAX
      MOVZX EAX,DATA.day
```

```
      PUSH  EAX
      PUSH OFFSET TIM
      PUSH OFFSET STRCOPY
      CALL wsprintfA
;отправить строку в окно редактирования
      PUSH OFFSET STRCOPY
      PUSH 0
      PUSH WM_SETTEXT
      PUSH DWORD PTR [EBP+08H]
      CALL SendMessageA@16
      JMP  LO ; бесконечный цикл
      POP  EBP
      RET  4
GETTIME ENDP
_TEXT ENDS
END START
```

Трансляция программы THREAD.ASM:

```
ML /c /coff thread.asm
RC thread.rc
LINK /SUBSYSTEM:WINDOWS thread.obj thread.res
```

Прокомментирую программу из листинга 3.2.2.

Прошу читателя взять на вооружение весьма полезную функцию `Sleep`. Эта функция особенно часто используется именно в потоках, дабы несколько высвободить процессорное время. Единственным параметром функции является минимальное количество миллисекунд, которые поток будет находиться в состоянии простоя. Почему минимальное? Дело в том, что данная функция заставляет поток отказаться от остатка кванта времени, который ему предназначен. Таким образом, планировщик вынужден будет передать управление другому потоку. Но вот вернется ли управление первому потоку сразу после истечения заданного интервала, в общем случае не очевидно. Если в качестве аргумента задать значение 0, то планировщик передаст управление другому потоку. Если потоков с таким же или большим приоритетом нет, то функция `Sleep` сразу возвратит управление.

Взаимодействие потоков

Поговорим теперь о многопотоковой программе. В принципе, если не предполагается, что потоки как-то взаимодействуют друг с другом, технически не имеет значения, запущен один или несколько потоков. Сложности возникают, когда работа одного потока зависит от деятельности другого потока. Здесь возможны самые разные ситуации. Давайте рассмотрим все по порядку.

Нам уже пришлось столкнуться с ситуацией, когда два параллельных процесса взаимодействовали друг с другом посредством глобальных переменных. Точнее, один процесс готовил данные для другого процесса. Здесь не было никакой сложности: просто один процесс с некоторой периодичностью менял содержимое переменной, а второй процесс, с другой периодичностью, читал из этой переменной. Если период обновления данных меньше периода получения данных, то мы почти с достоверностью (почти!) получаем, что второй процесс будет получать всегда свежие данные. Иногда соотношение между периодом обновления и периодом получения данных, как в нашем случае, вообще не имеет никакого значения.

Часто случается, что данные невозможно получать периодически. Они могут, например, зависеть от деятельности третьего процесса. Как же второй процесс узнает, что данные уже готовы для передачи? На первый взгляд проблема решается введением дополнительной переменной, назовем ее `FLAG`. Примем, что при `FLAG=0` данные не готовы, а при `FLAG=1` данные готовы. Далее действует весьма простая схема:

```
NO_DAT:
        ...
        CMP FLAG,1
        JNE NO_DAT
        ...  ;передача данных
        ...
        MOV FLAG,0
        ...
```

Это фрагмент, как вы понимаете, для второго потока. Первый же поток также должен проверять переменную `FLAG` и, если `FLAG=0`, поместить новые данные и установить значение переменной `FLAG`, равное единице. Данная схема совсем неплоха, например, когда один процесс ждет окончания работы другого процесса. Другими словами, данные являются результатом всей работы этого процесса. Например, запущен компилятор, а другой поток ждет окончания его работы, дабы вывести результаты этой работы на некое устройство. Эта ситуация весьма распространена, но все же случай этот частный.

А что будет, если процесс передачи данных должен производиться многократно? Легко видеть, что данная схема будет работать и в более сложном случае. Важно, однако, чтобы второй процесс менял содержимое `FLAG` только после того, как он возьмет данные, а первый процесс — после того, как положит данные. Если нарушить это правило, то может возникнуть коллизия, когда, например, второй процесс еще не взял данные, а они уже изменились.

Такой подход можно осуществить и в более общем случае, когда два потока (или два процесса) должны поочередно получать доступ к одному ресурсу.

Как легко видеть, данный подход предполагает, что ресурс открыт либо для одного, либо для другого процесса. Если поставить задачу несколько иным образом — процесс либо открыт, либо закрыт для доступа, то возникло бы некоторое затруднение. Действительно, вероятна такая ситуация, когда оба потока ожидают открытия ресурса. Другими словами, они непрерывно осуществляют проверку переменной `FLAG` (`CMP FLAG,1`). Может статься, что они оба почти одновременно обратятся к ресурсу. Совершенно ясно, что здесь возникает необходимость в "третьей силе", которая бы занималась распределением доступа к ресурсу. Например, посылала бы сообщение вначале одному потоку и, если он ожидает доступа, давала доступ именно ему, а затем подобный процесс повторяется со вторым потоком.

Схема, описанная выше, весьма хороша (см., однако, ниже), но при условии поочередного обращения к ресурсу. Если это в общем случае не выполняется, то данный подход может дать серьезный сбой. Все приведенные здесь рассуждения имеют одну цель — показать, что проблема взаимодействия потоков и процессов, их синхронизации, является и сложной, и актуальной для многозадачных операционных систем. Отсюда следует, что такие операционные системы должны иметь собственные средства синхронизации.[3]

> ***Замечание***
>
> Следует заметить, что непрерывный опрос в цикле некоторой переменной может взять на себя значительную часть машинного времени, а это в многозадачной среде совсем нехорошо.

Оставшаяся часть данного раздела будет всецело посвящена средствам синхронизации операционной системы Windows.

Семафоры

Семафор представляет собой глобальный объект, позволяющий синхронизировать работу двух или нескольких процессов или потоков. Для программиста семафор — это просто глобальный счетчик, но манипулировать им можно только с помощью специальных функций. Если счетчик равен *N*, это означает, что к ресурсу имеют доступ *N* процессов. Рассмотрим функции для работы с семафорами.

---

[3] К слову сказать, в однозадачной операционной системе MS-DOS проблема совместного функционирования резидентных программ стояла весьма остро. Несмотря на то, что программисты, их писавшие, добивались весьма значительных успехов, все же одновременная работа нескольких резидентных программ часто приводила к весьма заметным конфликтам.

`CreateSemaphor` — создает глобальный объект-семафор. Возвращает дескриптор семафора. Параметры функции:

- 1-й параметр — указатель на структуру, определяющую атрибуты доступа. Может иметь значение для семейства Windows NT. Обычно данный параметр полагается равным `NULL`;
- 2-й параметр — начальное значение счетчика семафора. Определяет, сколько задач имеют доступ к ресурсу вначале;
- 3-й параметр — количество задач, которые имеют одновременный доступ к ресурсу;
- 4-й параметр — указатель на строку, содержащую имя семафора.

`OpenSemaphor` — открыть уже созданный семафор. Возвращает дескриптор семафора. Данную функцию используют не так часто. Обычно создают семафор и присваивают его дескриптор глобальной переменной, а потом используют этот дескриптор в порождаемых потоках. Единственный параметр функции определяет желаемый уровень доступа к семафору. Возможные значения:

- `SEMAPHORE_MODIFY_STATE=2H` — разрешить использование функции `ReleaseSemaphore`;
- `SYNCHRONIZE=100000H` — разрешить использование любой функции ожидания, только для семейства Windows NT;
- `SEMAPHORE_ALL_ACCESS=0F0000h+SYNCHRONIZE+3H` — специфицирует все возможные флаги доступа к семафору.

`WaitForSingleObject` — ожидает открытие семафора. При успешном завершении, т. е. открытии доступа к объекту, функция возвращает 0. Значение `102h` будет означать, что закончился заданный период ожидания. Параметры функции:

- 1-й параметр — дескриптор семафора;
- 2-й параметр — время ожидания в миллисекундах. Если параметр равен `INFINITE = 0FFFFFFFFh`, то время ожидания не ограничено.

`ReleaseSemaphor` — освободить семафор и тем самым позволить получить доступ к ресурсу другим процессам. Параметры функции:

- 1-й параметр — дескриптор семафора;
- 2-й параметр определяет, какое значение должно быть добавлено к счетчику семафора. Чаще всего этот параметр равен единице;
- 3-й параметр — указатель на переменную, куда должно быть помещено предыдущее значение счетчика.

Рассмотрим алгоритм работы с семафором. Сначала при помощи функции `CreateSemaphor` создается семафор, и его дескриптор присваивается глобальной переменной. Перед попыткой обращения к ресурсам, доступ к которым необходимо ограничить, поток должен вызвать функцию `WaitForSingleObject`. При открытии доступа функция возвращает 0. По окончании работы с ресурсом следует вызвать функцию `ReleaseSemaphor`. Тем самым увеличивается счетчик доступа на 1, в свою очередь функция `WaitForSingleObject` уменьшает значение счетчика. С помощью семафора можно регулировать количество потоков, которые одновременно могут иметь доступ к ресурсу. Максимальное значение счетчика как раз и определяет, сколько потоков могут получить доступ к ресурсу одновременно. Но обычно, как я уже говорил, максимальное значение полагают равным 1.

События

Событие является объектом, очень похожим на семафор, но в несколько видоизмененном виде. Рассмотрим функции для работы с событиями.

`CreateEvent` — создает объект — событие. Параметры функции:

- 1-й параметр имеет тот же смысл, что и первый параметр функции `CreateSemaphor`. Обычно полагается равным `NULL`;
- 2-параметр — если параметр не равен нулю, то событие может быть сброшено при помощи функции `ResetEvent`. Иначе событие сбрасывается, когда к нему осуществляется доступ другого процесса;
- 3-й параметр — если параметр равен 0, то событие инициализируется как сброшенное, в противном случае сразу же подается сигнал о наступлении соответствующей ситуации;
- 4-й параметр — указатель на строку, которая содержит имя события.

Ожидание события осуществляется, как и в случае с семафором, функцией `WaitForSingleObject`.

Функция `OpenEvent` аналогична функции `OpenSemaphor`, и на ней мы останавливаться не будем.

`SetEvent` — подать сигнал о наступлении события. Единственный параметр функции — дескриптор события.

Критические секции

Критическая секция позволяет уберечь определенные области программы так, чтобы в этой области программы в данный момент времени исполнялся бы только один поток. Рассмотрим функции для работы с критической секцией.

Функция `InitializeCriticalSection` создает объект под названием "критическая секция". Параметр функции — указатель на структуру, указанную ниже. Поля данной структуры используются только внутренними процедурами, и смысл их безразличен.

```
CRITICAL_SECTION STRUCT
      DebugInfo      DWORD  ?
      LockCount      LONG   ?
      RecursionCount LONG   ?
      OwningThread   HANDLE ?
      LockSemaphore  HANDLE ?
      SpinCount      DWORD  ?
CRITICAL_SECTION ENDS
```

`EnterCriticalSection` — войти в критическую секцию. После выполнения этой функции данный поток становится владельцем данной секции. Следующий поток, вызвав данную функцию, будет находиться в состоянии ожидания. Параметр функции такой же, что и в предыдущей функции.

`LeaveCriticalSection` — покинуть критическую секцию. После этого второй поток, который был остановлен функцией `EnterCriticalSection`, станет владельцем критической секции. Параметр функции `LeaveCriticalSection` такой же, как и у предыдущих функций.

`DeleteCriticalSection` — удалить объект "критическая секция". Параметр аналогичен предыдущим.

Программно можно определить несколько объектов критической секции, с которыми будут работать несколько потоков. Мы не зря, говоря о критических секциях, упоминаем только потоки. Разные процессы (точнее, потоки в разных процессах) не могут использовать данный объект синхронизации.

Теперь рассмотрим пример использования критической секции. Примеры использования семафоров и событий вы сможете найти в книге [4]. Изложим вкратце идею, положенную в основу примера из листинга 3.2.3. Два потока время от времени обращаются к процедуре, выводящей очередной символ из строки в окно. В результате такой конкурентной деятельности должна быть напечатана строка. Часть процедуры, выводящей очередной символ, сделана критической, поэтому доступ к выводу в окно в данный момент времени имеет только один поток.

Листинг 3.2.3. Пример синхронизации двух потоков посредством критической секции

```
;файл thread.inc
;константы
;сообщение приходит при закрытии окна
```

```
WM_DESTROY      equ 2
;сообщение приходит при создании окна
WM_CREATE       equ 1
;сообщение при щелчке левой кнопкой мыши в области окна
WM_LBUTTONDOWN  equ 201h
;сообщение при щелчке правой кнопкой мыши в области окна
WM_RBUTTONDOWN  equ 204h
;свойства окна
CS_VREDRAW      equ 1h
CS_HREDRAW      equ 2h
CS_GLOBALCLASS  equ 4000h
WS_OVERLAPPEDWINDOW equ 000CF0000H
stylcl  equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
DX0    equ 300
DY0    equ 200
;компоненты цветов
RED    equ 50
GREEN  equ 50
BLUE   equ 255
RGBW   equ (RED or (GREEN shl 8)) or (BLUE shl 16)
RGBT   equ 255 ;красный
;идентификатор стандартной пиктограммы
IDI_APPLICATION  equ 32512
;идентификатор курсора
IDC_CROSS        equ 32515
;режим показа окна - нормальный
SW_SHOWNORMAL    equ 1
;прототипы внешних процедур
EXTERN  Sleep@4:NEAR
EXTERN  CreateThread@24:NEAR
EXTERN  InitializeCriticalSection@4:NEAR
EXTERN  EnterCriticalSection@4:NEAR
EXTERN  LeaveCriticalSection@4:NEAR
EXTERN  DeleteCriticalSection@4:NEAR
EXTERN  GetTextExtentPoint32A@16:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
```

```
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
EXTERN  TextOutA@20:NEAR
EXTERN  CreateSolidBrush@4:NEAR
EXTERN  SetBkColor@8:NEAR
EXTERN  SetTextColor@8:NEAR
EXTERN  GetDC@4:NEAR
EXTERN  DeleteDC@4:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND    DD ?
      MSMESSAGE DD ?
      MSWPARAM  DD ?
      MSLPARAM  DD ?
      MSTIME    DD ?
      MSPT      DD ?
MSGSTRUCT ENDS
;---------
WNDCLASS STRUC
      CLSSTYLE        DD ?
      CLSLPFNWNDPROC  DD ?
      CLSCBCLSEXTRA   DD ?
      CLSCBWNDEXTRA   DD ?
      CLSHINSTANCE    DD ?
      CLSHICON        DD ?
      CLSHCURSOR      DD ?
      CLSHBRBACKGROUND DD ?
      MENNAME         DD ?
      CLSNAME         DD ?
WNDCLASS ENDS
;структура для работы с критической секцией
CRIT STRUC
      DD ?
      DD ?
      DD ?
      DD ?
      DD ?
      DD ?
CRIT ENDS
;структура для определения длины текста
SIZET STRUC
      X1  DWORD ?
      Y1  DWORD ?
SIZET ENDS
;файл thread.asm
```

```
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;-------------------------------------------------
include thread.inc
;подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\gdi32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      NEWHWND   DD 0
      MSG       MSGSTRUCT <?>
      WC        WNDCLASS  <?>
      SZT       SIZET   <?>
      HINST     DD 0
      TITLENAME DB 'Вывод в окно двумя потоками',0
      NAM       DB 'CLASS32',0
      XT        DD 30
      YT        DD 30
      HW        DD ?
      DC        DD ?
      TEXT      DB 'Текст в окне красный',0
      SPA       DB '            '
                  DB '            ',0
      IND       DD 0
      SK        CRIT <?>
      THR1      DD ?
      THR2      DD ?
      FLAG1     DD 0
      FLAG2     DD 0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV  [HINST], EAX
REG_CLASS:
;заполнить структуру окна
;стиль
      MOV  [WC.CLSSTYLE],stylcl
;процедура обработки сообщений
      MOV  [WC.CLSLPFNWNDPROC],OFFSET WNDPROC
```

```
      MOV  [WC.CLSCBCLSEXTRA],0
      MOV  [WC.CLSCBWNDEXTRA],0
      MOV  EAX,[HINST]
      MOV  [WC.CLSHINSTANCE],EAX
;----------пиктограмма окна
      PUSH  IDI_APPLICATION
      PUSH  0
      CALL  LoadIconA@8
      MOV  [WC.CLSHICON], EAX
;----------курсор окна
      PUSH  IDC_CROSS
      PUSH  0
      CALL  LoadCursorA@8
      MOV  [WC.CLSHCURSOR], EAX
;----------
      PUSH  RGBW                 ; цвет кисти
      CALL  CreateSolidBrush@4   ; создать кисть
      MOV  [WC.CLSHBRBACKGROUND],EAX
      MOV   DWORD PTR [WC.MENNAME],0
      MOV   DWORD PTR [WC.CLSNAME],OFFSET NAM
      PUSH  OFFSET WC
      CALL  RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH  0
      PUSH  [HINST]
      PUSH  0
      PUSH  0
      PUSH  DY0      ; DY0 - высота окна
      PUSH  DX0      ; DX0 - ширина окна
      PUSH  100      ; координата Y
      PUSH  100      ; координата X
      PUSH  WS_OVERLAPPEDWINDOW
      PUSH  OFFSET TITLENAME   ; имя окна
      PUSH  OFFSET NAM         ; имя класса
      PUSH  0
      CALL  CreateWindowExA@48
;проверка на ошибку
      CMP   EAX,0
      JZ   _ERR
      MOV  [NEWHWND], EAX      ; дескриптор окна
;-----------------------------------
      PUSH  SW_SHOWNORMAL
      PUSH  [NEWHWND]
      CALL  ShowWindow@8       ; показать созданное окно
;-----------------------------------
      PUSH  [NEWHWND]
```

```
      CALL  UpdateWindow@4     ; перерисовать видимую часть окна
;цикл обработки сообщений
MSG_LOOP:
      PUSH  0
      PUSH  0
      PUSH  0
      PUSH  OFFSET MSG
      CALL  GetMessageA@16
      CMP   AX, 0
      JE    END_LOOP
      PUSH  OFFSET MSG
      CALL  TranslateMessage@4
      PUSH  OFFSET MSG
      CALL  DispatchMessageA@4
      JMP   MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
      PUSH  [MSG.MSWPARAM]
      CALL  ExitProcess@4
_ERR:
      JMP   END_LOOP
;-----------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
      CMP  DWORD PTR [EBP+0CH],WM_DESTROY
      JE   WMDESTROY
      CMP  DWORD PTR [EBP+0CH],WM_CREATE
      JE   WMCREATE
      CMP  DWORD PTR [EBP+0CH],WM_LBUTTONDOWN
      JNE  CONTIN
;проверить флаг запуска
      CMP  FLAG1,0
      JNE  DEFWNDPROC
      MOV  FLAG1,1
;инициализировать указатели
      LEA  EAX,TEXT
```

```
      MOV  IND,EAX
      MOV  XT,30
;запуск первого потока
      PUSH OFFSET THR1
      PUSH 0
      PUSH EAX
      PUSH OFFSET THREAD1
      PUSH 0
      PUSH 0
      CALL CreateThread@24
;запуск второго потока
      PUSH OFFSET THR2
      PUSH 0
      PUSH EAX
      PUSH OFFSET THREAD2
      PUSH 0
      PUSH 0
      CALL CreateThread@24
      JMP  DEFWNDPROC
CONTIN:
      CMP  DWORD PTR [EBP+0CH],WM_RBUTTONDOWN
      JNE  DEFWNDPROC
;проверить флаг запуска
      CMP  FLAG2,0
      JNE  DEFWNDPROC
      MOV  FLAG2,1
;инициализировать указатели
      LEA  EAX,SPA
      MOV  IND,EAX
      MOV  XT,30
;запуск первого потока
      PUSH OFFSET THR1
      PUSH 0
      PUSH EAX
      PUSH OFFSET THREAD1
      PUSH 0
      PUSH 0
      CALL CreateThread@24
;запуск второго потока
      PUSH OFFSET THR2
      PUSH 0
      PUSH EAX
      PUSH OFFSET THREAD2
      PUSH 0
      PUSH 0
      CALL CreateThread@24
```

```
      JMP  DEFWNDPROC
WMCREATE:
      MOV  EAX,DWORD PTR [EBP+08H]
;запомнить дескриптор окна в глобальной переменной
      MOV  HW,EAX
;инициализировать критическую секцию
      PUSH OFFSET SK
      CALL InitializeCriticalSection@4
      MOV  EAX, 0
      JMP  FINISH
DEFWNDPROC:
      PUSH  DWORD PTR [EBP+14H]
      PUSH  DWORD PTR [EBP+10H]
      PUSH  DWORD PTR [EBP+0CH]
      PUSH  DWORD PTR [EBP+08H]
      CALL  DefWindowProcA@16
      JMP   FINISH
WMDESTROY:
;удалить критическую секцию
      PUSH  OFFSET SK
      CALL  DeleteCriticalSection@4
      PUSH  0
      CALL  PostQuitMessage@4 ;WM_QUIT
      MOV   EAX, 0
FINISH:
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      RET 16
WNDPROC  ENDP
;вывод
OUTSTR PROC
;проверяем, не закончился ли текст
      MOV EBX,IND
      CMP BYTE PTR [EBX],0
      JNE NO_0
      RET
NO_0:
;вход в критическую секцию
      PUSH OFFSET SK
      CALL EnterCriticalSection@4
;----------------
      PUSH HW
      CALL GetDC@4
      MOV DC,EAX
```

```
;---------------- цвет фона = цвет окна
      PUSH RGBW
      PUSH EAX
      CALL SetBkColor@8
;---------------- цвет текста (красный)
      PUSH RGBT
      PUSH DC
      CALL SetTextColor@8
;---------------- вывести текст
      PUSH 1
      PUSH IND
      PUSH YT
      PUSH XT
      PUSH DC
      CALL TextOutA@20
;- вычислить длину текста в пикселах текста
      PUSH OFFSET SZT
      PUSH 1
      PUSH IND
      PUSH DC
      CALL GetTextExtentPoint32A@16
;увеличить указатели
      MOV EAX,SZT.X1
      ADD XT,EAX
      INC IND
;---------------- закрыть контекст
      PUSH DC
      CALL DeleteDC@4
;выход из критической секции
      PUSH OFFSET SK
      CALL LeaveCriticalSection@4
      RET
OUTSTR ENDP
;первый поток
THREAD1 PROC
LO1:
;проверить, не конец ли текста
      MOV  EBX,IND
      CMP  BYTE PTR [EBX],0
      JE   _END1
;вывод очередного символа
      CALL OUTSTR
;задержка
      PUSH 1000
      CALL Sleep@4
      JMP  LO1
```

```
_END1:
      RET 4
THREAD1 ENDP
;второй поток
THREAD2 PROC
LO2:
;проверить, не конец ли текста
      MOV  EBX,IND
      CMP  BYTE PTR [EBX],0
      JE   _END2
;вывод очередного символа
      CALL OUTSTR
;задержка
      PUSH 1000
      CALL Sleep@4
      JMP  LO2
_END2:
      RET  4
THREAD2 ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.2.3:

```
ML /c /coff /DMASM thread.asm
LINK /SUBSYSTEM:WINDOWS thread.obj
```

А теперь комментарий к программе.

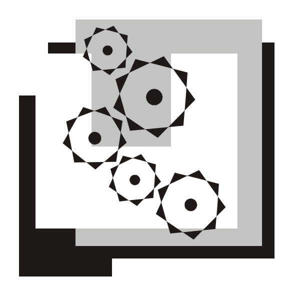
При нажатии левой кнопки мыши начинается вывод текстовой строки. При нажатии правой кнопки мыши выведенная строка стирается. Флаги `FLAG1` и `FLAG2` нужны для того, чтобы вывод строки и вывод пустой строки можно было производить только один раз. Для того чтобы несколько замедлить вывод текста, мы вводим задержку (`Sleep`) в цикл вызова процедуры `OUTSTR` в каждом потоке. Обратите внимание, что буквы выводятся в окно в основном парами. Объясняется это тем, что пока один из потоков выводит символ, второй уже ждет разрешения, и, как только первый поток выходит из критической секции, второй поток сразу выводит следующий символ. После в обоих потоках срабатывает задержка (функция `Sleep`). Ради эксперимента удалите из программы критическую секцию. В результате с большой долей вероятности строка будет выведена с ошибкой.

Завершая разговор о критических секциях, отмечу, что это наиболее быстрый способ синхронизации. К недостаткам данного подхода относится невозможность доступа к секции сразу нескольких потоков, а также отсутствие специальных средств, чтобы подсчитывать число обращений к ресурсу.

Взаимоисключения

Я не упомянул еще один способ синхронизации, отложив его подробное описание на следующие главы. Здесь же отмечу, что этот способ называется "взаимоисключением" или мьютексом (mutex). Данный способ синхронизации не удобен для работы с потоками, он более пригоден для процессов. Данный объект создается при помощи функции `CreateMutex`. Все процессы, пытающиеся создать уже созданный объект, получают дескриптор уже существующего, созданного другим процессом объекта "взаимоисключение". Особенность данного объекта, прежде всего, в том, что им может владеть только один процесс. В документации фирмы Microsoft рекомендуется использовать этот объект для определения, запущено уже данное приложение или нет. Но об этом речь пойдет далее *(см. главу 3.5)*.

Глава 3.3



Создание динамических библиотек

Мне, воспитанному на операционной системе MS-DOS, название "динамическая библиотека" в начале резало слух. В моем понимании библиотека должна была подключаться на стадии компоновки. Что касается динамических библиотек, то это очень напоминало такое понятие, как оверлей. В MS-DOS оверлеи использовались для того, чтобы сэкономить оперативную память. Менеджер оверлеев загружал отдельные части оверлея, выгружал другие. Экономии памяти можно было достигнуть колоссальной, а в MS-DOS это было самым слабым местом. Однако оверлей использовался только одним приложением. Динамическая же библиотека может использоваться несколькими приложениями одновременно — отсюда и "библиотека" (публичное заведение). Заметим, что все API-функции, которые мы используем в своих программах, также реализованы в виде динамических библиотек, которые автоматически подключаются при запуске исполняемых модулей. Динамические библиотеки можно создавать на разных языках программирования, тем самым реально осуществляя интегрирование разных алгоритмических языков.

Общие понятия

Использование динамических библиотек (по-другому — библиотек динамической компоновки) — это способ осуществления модульности в период выполнения программы. Динамическая библиотека (Dynamic Link Library, DLL) позволяет упростить и саму разработку программного обеспечения. Вместо того чтобы каждый раз перекомпилировать огромные exe-программы, достаточно перекомпилировать лишь отдельный динамический модуль. Кроме того, доступ к динамической библиотеке возможен сразу из нескольких исполняемых модулей, что делает многозадачность более гибкой. При обращении к одной и той же динамической библиотеке несколькими исполняемыми

приложениями в памяти содержится лишь один экземпляр динамической библиотеки. С другой стороны, динамическая библиотека содержится (проецируется) в том же адресном пространстве, что и использующий их процесс. Можно сказать, что код динамической библиотеки становится частью кода данного процесса и доступен из любого потока этого процесса. Все, что создается в функциях динамической библиотеки, автоматически принадлежит и основному процессу.

Структура исполняемых модулей будет рассмотрена в *приложении 4*, но главное то, что структура эта практически такая же, как и у exe-модуля. Тот, кто программировал под MS-DOS, должен быть знаком с понятием оверлея. По своей функциональности динамическая библиотека очень похожа на оверлей, но название "динамическая библиотека" более удачно[1].

При написании exe-модулей вы уже познакомились с тем, как определять импортируемые функции. Достаточно объявить эти функции как `EXTERN`. При создании динамической библиотеки вам придется указывать и импортируемые, и экспортируемые функции.

Для того чтобы двигаться дальше, введу такое понятие, как связывание. Собственно, мы уже познакомились с этим понятием, когда рассматривали работу редактора связей (см. *главу 1.1*, рис. 1.1.1 и комментарий к нему). Во время трансляции связываются имена, указанные в программе как внешние (`EXTERN`), с соответствующими именами из библиотек, которые указываются при помощи директивы `INCLUDELIB`. Такое связывание называется *ранним* (или статическим). Напротив, в случае с динамической библиотекой связывание происходит во время выполнения модуля. Это связывание называют *поздним* (или динамическим). При этом позднее связывание может происходить в автоматическом режиме в начале запуска программы и при помощи специальных API-функций (см. листинг 3.3.2), по желанию программиста. При этом говорят о явном и неявном связывании. Сказанное иллюстрирует рис. 3.3.1. Замечу также, что использование динамической библиотеки экономит дисковое пространство, т. к. представленная в библиотеке процедура содержится лишь один раз, в отличие от процедур, помещаемых в модули из статических библиотек.[2]

---

[1] Оверлей (Overlay) в переводе означает "перекрытие" — указание на то, что в оверлейную область памяти могут загружаться по очереди разные части оверлея, перекрывая друг друга.

[2] Вообще говоря, библиотеки, используемые нами для программирования в Windows, такие как user32.lib, kernel32.lib и т. п., правильнее называть не статическими библиотеками, а библиотеками импорта. В них нет программного кода, а лишь информация, используемая для трансляции.

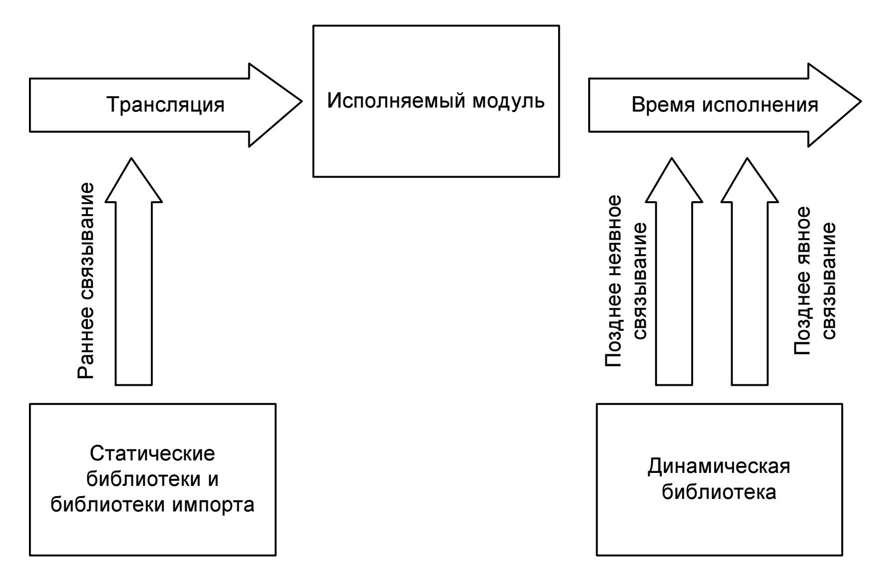


Рис. 3.3.1. Иллюстрация понятия "связывание"

В среде Windows практикуются два механизма связывания: по символьным именам и по порядковым номерам. В первом случае функция, определенная в динамической библиотеке, идентифицируется по имени, во втором — по порядковому номеру, который должен быть задан при трансляции. Связывание по порядковому номеру в основном практиковалось в старой операционной системе Windows 3.*x*. На мой взгляд, связывание по имени — более удобный механизм.

Динамическая библиотека может содержать также ресурсы. Так, файлы шрифтов представляют собой динамические библиотеки, единственным содержимым которых являются ресурсы. Я уже отметил, что динамическая библиотека становится как бы продолжением вашей программы, загружаясь в адресное пространство процесса. Соответственно, данные процесса доступны из динамической библиотеки, и, наоборот, данные динамической библиотеки доступны для процесса.

В любой динамической библиотеке следует определить точку входа (процедура входа). По умолчанию за точку входа принимают метку, указываемую за директивой `END` (например, `END START`). При загрузке динамической библиотеки и выгрузке динамической библиотеки автоматически вызывается процедура входа. Замечу при этом, что каким бы способом ни была загружена динамическая библиотека (явно или неявно), выгрузка динамической библиотеки из памяти будет происходить автоматически при закрытии

процесса или потока. В принципе, процедура входа может быть использована для некоторой начальной инициализации переменных. Довольно часто эта процедура остается пустой. При вызове процедуры входа в нее помещаются три параметра:

- 1-й параметр — идентификатор DLL-модуля;
- 2-й параметр — причина вызова *(см. далее)*;
- 3-й параметр — резерв.

Рассмотрим подробнее второй параметр процедуры входа. Вот четыре возможных значения этого параметра:

```
DLL_PROCESS_DETACH  equ 0
DLL_PROCESS_ATTACH  equ 1
DLL_THREAD_ATTACH   equ 2
DLL_THREAD_DETACH   equ 3
```

Здесь:

- `DLL_PROCESS_ATTACH` сообщает, что динамическая библиотека загружена в адресное пространство вызывающего процесса;
- `DLL_THREAD_ATTACH` сообщает, что текущий процесс создает новый поток. Такое сообщение посылается всем динамическим библиотекам, загруженным к этому времени процессом;
- `DLL_PROCESS_DETACH` сообщает, что динамическая библиотека выгружается из адресного пространства процесса;
- `DLL_THREAD_DETACH` сообщает, что некий поток, созданный данным процессом, в адресное пространство которого загружена данная динамическая библиотека, уничтожается.

Создание динамических библиотек

Перейдем теперь к разбору программных примеров динамических библиотек. В листинге 3.3.1 приводится пример простейшей динамической библиотеки. Данная динамическая библиотека, по сути, ничего не делает. Просто при загрузке библиотеки, при ее выгрузке, а также при вызове процедуры `DLLP1`, которая в библиотеке содержится, будет выдано обычное Windows-сообщение. Обратите внимание, как определяется процесс загрузки и выгрузки библиотеки. Замечу также, что процедура входа должна возвращать ненулевое значение. Процедура `DLLP1` обрабатывает также один параметр, передаваемый через стек обычным способом.

Листинг 3.3.1. Простейшая DLL-библиотека

```
.586P
;плоская модель памяти
.MODEL FLAT,  stdcall
PUBLIC DLLP1
;константы
;сообщения, приходящие при открытии
;динамической библиотеки
DLL_PROCESS_DETACH   equ 0
DLL_PROCESS_ATTACH   equ 1
DLL_THREAD_ATTACH    equ 2
DLL_THREAD_DETACH    equ 3
;прототипы внешних процедур
EXTERN      MessageBoxA@16:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;----------------------------------------------
;сегмент данных
_DATA SEGMENT
      TEXT1 DB 'Вход в библиотеку',0
      TEXT2 DB 'Выход из библиотеки',0
      MS    DB 'Сообщение из библиотеки',0
      TEXT  DB 'Вызов процедуры из DLL',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
; [EBP+10H]   ;резервный параметр
; [EBP+0CH]   ;причина вызова
; [EBP+8]     ;идентификатор DLL-модуля
DLLENTRY  PROC
      PUSH EBP
      MOV  EBP,ESP
      MOV  EAX,DWORD PTR [EBP+0CH]
      CMP  EAX,0
      JNE  D1
;закрытие библиотеки
      PUSH 0
      PUSH OFFSET MS
      PUSH OFFSET TEXT2
      PUSH 0
      CALL MessageBoxA@16
      JMP _EXIT
D1:
      CMP  EAX,1
      JNE _EXIT
```

```
;открытие библиотеки
      PUSH 0
      PUSH OFFSET MS
      PUSH OFFSET TEXT1
      PUSH 0
      CALL MessageBoxA@16
_EXIT:
      MOV  EAX,1
      LEAVE
      RET  12
DLLENTRY ENDP
;--------------------
; [EBP+8]    ;параметр процедуры
DLLP1 PROC EXPORT
      PUSH  EBP
      MOV   EBP,ESP
      CMP   DWORD PTR [EBP+8],1
      JNE   _EX
      PUSH  0
      PUSH  OFFSET MS
      PUSH  OFFSET TEXT
      PUSH  0
      CALL  MessageBoxA@16
_EX:
      POP   EBP
      RET   4
DLLP1 ENDP
_TEXT ENDS
END DLLENTRY
```

Трансляция программы из листинга 3.3.1:

```
ml /c /coff  dll1.asm
link /subsystem:windows /DLL /ENTRY:DLLENTRY dll1.obj
```

Прежде всего, обратите внимание, что после процедуры, вызываемой из другого модуля, мы указали ключевое слово `EXPORT`. Это слово необходимо для правильной трансляции. Процедура `DLLP1` должна быть определена как `PUBLIC`. Для создания динамических библиотек в командной строке `link` следует указать ключ `/DLL`. Ключ `/ENTRY:DLLENTRY` в строке `link` можно опустить, т. к. точка входа определяется из директивы `END DLLENTRY`. Наконец, внимательно посмотрите на структуру процедуры входа. Поскольку она всегда получает три параметра, мы должны освобождать стек при выходе из нее (`RET 12`).

В листинге 3.3.2 представлена программа, которая загружает динамическую библиотеку, показанную в листинге 3.3.1. Это пример позднего явного свя-

зывания. Библиотека должна быть вначале загружена при помощи функции `LoadLibrary`. Затем определяется адрес процедуры с помощью функции `GetProcAddress`, после чего можно осуществлять вызов. Как и следовало ожидать, MASM32 помещает в динамическую библиотеку вместо `DLLP1` имя `_DLLP1@0`. Это мы учитываем в нашей программе. Мы учитываем также возможность ошибки при вызове функций `LoadLibrary` и `GetProcAddress`. В этой связи укажем, как (в какой последовательности) ищет библиотеку функция `LoadLibrary`:

- поиск в каталоге, откуда была запущена программа;
- поиск в текущем каталоге;
- в системном каталоге (получить можно с помощью функции `GetSystemDirectory`);
- в каталоге Windows (получить можно с помощью функции `GetWindowsDirectory`);
- в каталогах, указанных в окружении (`PATH`).

В конце программы мы выгружаем из памяти динамическую библиотеку, что, кстати, могли бы и не делать, т. к. при выходе из программы эта процедура выполняется автоматически.

Листинг 3.3.2. Вызов динамической библиотеки (листинг 3.3.1). Явное связывание

```
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
;прототипы внешних процедур
EXTERN  GetProcAddress@8:NEAR
EXTERN  LoadLibraryA@4:NEAR
EXTERN  FreeLibrary@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  MessageBoxA@16:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;----------------------------------------
;сегмент данных
_DATA SEGMENT
      TXT      DB 'Ошибка динамической библиотеки',0
      MS       DB 'Сообщение',0
      LIBR     DB 'DLL1.DLL',0
      HLIB     DD ?
```

```
      NAMEPROC DB '_DLLP1@0',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;загрузить библиотеку
      PUSH OFFSET LIBR
      CALL LoadLibraryA@4
      CMP  EAX,0
      JE  _ERR
      MOV  HLIB,EAX
;получить адрес процедуры
      PUSH OFFSET NAMEPROC
      PUSH HLIB
      CALL GetProcAddress@8
      CMP  EAX,0
      JNE  YES_NAME
;сообщение об ошибке
_ERR:
      PUSH  0
      PUSH  OFFSET MS
      PUSH  OFFSET TXT
      PUSH  0
      CALL  MessageBoxA@16
      JMP  _EXIT
YES_NAME:
      PUSH 1 ; параметр
      CALL EAX
;закрыть библиотеку
      PUSH HLIB
      CALL FreeLibrary@4
;библиотека автоматически закрывается также
;при выходе из программы
;выход
_EXIT:
      PUSH 0
      CALL ExitProcess@4
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.3.2 ничем не отличается от трансляции обычных программ:

```
ml /c /coff dllex.asm
link /subsystem:windows dllex.obj
```

Неявное связывание

Хотя я и указал, что, на мой взгляд, неявное связывание менее гибко, я нашел необходимым привести пример неявного связывания. Тем более, что все системные динамические библиотеки подключаются нами именно неявно.

Мы рассмотрим здесь только вызывающую программу, т. к. вызываемая программа, естественно, не меняется. Как видите, текст программы стал несколько проще (см. листинг 3.3.3). Здесь важно заметить, что, во-первых, необходимо объявить вызываемую из динамической библиотеки процедуру как внешнюю, а во-вторых, подключить статическую библиотеку DLLP1.LIB, которая автоматически создается при компоновке динамической библиотеки, если имеется хотя бы одно имя, к которому открывает доступ динамическая библиотека.

Листинг 3.3.3. Вызов динамической библиотеки. Неявное связывание

```
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
;прототипы внешних процедур
EXTERN      DLLP1@0:NEAR
EXTERN      ExitProcess@4:NEAR
;директивы компоновщику для подключения библиотек
includelib dll1.lib
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
      PUSH 1 ;параметр
      CALL DLLP1@0
;выход
_EXIT:
      PUSH 0
      CALL ExitProcess@4
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.3.3:

```
ml /c /coff dllex.asm
link /subsystem:windows dllex.obj
```

У читателя, скорее всего, возникнет вопрос, откуда появляется библиотека DLLP1.LIB? Здесь все достаточно просто. Транслятор MASM32, как я уже сказал, создает библиотеку автоматически. Единственным условием создания является наличие в модулях, из которых создается библиотека, имен (процедур или переменных), которые будут предоставляться для доступа извне (имена, определяемые с помощью модификатора `PUBLIC`).

Ранее было сказано, что возможным механизмом связывания является определение адреса процедуры через порядковый номер. Изложу схему того, как это можно сделать. Сначала вы должны сопоставить процедуре, которая будет вызываться из динамической библиотеки, некое двухбайтное число. Это делается посредством строки в def-файле: `EXPORTS DLLP1 @1`. Здесь процедуре `DLLP1` сопоставляется номер 1. Для того чтобы использовать def-файл при компоновке, следует в командной строке указать ключ `/DEF:`*имя_файла*. После это производится трансляция, и динамическая библиотека готова. Теперь при вызове функции `GetProcAddress` вторым параметром следует указать порядковый номер, точнее, двойное слово, младшее слово которого есть порядковый номер, а старшее слово равно нулю. И все будет работать точно так же, как и раньше. Лично я не вижу особой необходимости использовать такой подход: при замене старых динамических библиотек на новые — имена, как правило, остаются теми же для совместимости, а номера могут измениться.

Использование общего адресного пространства

В листинге 3.3.4 представлен текст динамической библиотеки и программы, вызывающей процедуру из этой библиотеки. Здесь ничего сложного, просто я хотел продемонстрировать, что основной процесс и динамическая библиотека используют одно и то же адресное пространство. Процесс передает адреса строк, которые находятся в блоке данных основного процесса. В свою очередь, процедура возвращает в основной процесс адрес строки, находящейся в блоке данных динамической библиотеки.

Листинг 3.3.4. Основной модуль и динамическая библиотека. Передача параметров

```
;динамическая библиотека DLL2.ASM
.586P
;плоская модель памяти
```

```
.MODEL FLAT, stdcall
PUBLIC DLLP1
;константы
;сообщения, приходящие при открытии динамической библиотеки
DLL_PROCESS_DETACH  equ 0
DLL_PROCESS_ATTACH  equ 1
DLL_THREAD_ATTACH   equ 2
DLL_THREAD_DETACH   equ 3
;прототипы внешних процедур
EXTERN  MessageBoxA@16:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      TEXT DB "Строка в динамической библиотеке",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
; [EBP+10H]   ;резервный параметр
; [EBP+0CH]   ;причина вызова
; [EBP+8]     ;идентификатор DLL-модуля
DLLENTRY PROC
      PUSH EBP
      MOV EBP,ESP
      MOV EAX,DWORD PTR [EBP+0CH]
      CMP EAX,0
      JNE D1
;закрытие библиотеки
      JMP _EXIT
D1:
      CMP EAX,1
      JNE _EXIT
;открытие библиотеки
_EXIT:
      MOV EAX,1
      LEAVE
      RET 12
DLLENTRY ENDP
;--------------------
;адреса параметров
; [EBP+8]
; [EBP+0CH]
DLLP1 PROC EXPORT
      PUSH  EBP
```

```
      MOV   EBP,ESP
      PUSH  0
      PUSH  DWORD PTR [EBP+0CH]
      PUSH  DWORD PTR [EBP+8]
      PUSH  0
      CALL  MessageBoxA@16
      POP   EBP
      LEA   EAX,TEXT
      RET   8
DLLP1 ENDP
_TEXT ENDS
END DLLENTRY
;основной модуль DLLEX2.ASM, вызывающий
;процедуру из динамической библиотеки
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
;прототипы внешних процедур
EXTERN  GetProcAddress@8:NEAR
EXTERN  LoadLibraryA@4:NEAR
EXTERN  FreeLibrary@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  MessageBoxA@16:NEAR
;---------------------------------------------
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;---------------------------------------------
;сегмент данных
_DATA SEGMENT
      TXT  DB 'Ошибка динамической библиотеки',0
      MS   DB 'Сообщение',0
      LIBR DB 'DLL2.DLL',0
      HLIB DD ?
      MS1  DB 'Сообщение из библиотеки',0
      TEXT DB 'Строка содержится в основном модуле',0
      NAMEPROC DB '_DLLP1@0',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
; [EBP+10H]   ;резервный параметр
; [EBP+0CH]   ;причина вызова
; [EBP+8]     ;идентификатор DLL-модуля
START:
;загрузить библиотеку
      PUSH OFFSET LIBR
```

```
      CALL LoadLibraryA@4
      CMP  EAX,0
      JE  _ERR
      MOV  HLIB,EAX
;получить адрес
      PUSH OFFSET NAMEPROC
      PUSH HLIB
      CALL GetProcAddress@8
      CMP  EAX,0
      JNE  YES_NAME
;сообщение об ошибке
_ERR:
      PUSH  0
      PUSH  OFFSET MS
      PUSH  OFFSET TXT
      PUSH  0
      CALL  MessageBoxA@16
      JMP  _EXIT
YES_NAME:
      PUSH  OFFSET MS1
      PUSH  OFFSET TEXT
      CALL  EAX
      PUSH  0
      PUSH  OFFSET MS
      PUSH  EAX
      PUSH  0
      CALL  MessageBoxA@16
;закрыть библиотеку
      PUSH  HLIB
      CALL  FreeLibrary@4
;библиотека автоматически закрывается также
;при выходе из программы
;выход
_EXIT:
      PUSH 0
      CALL ExitProcess@4
_TEXT ENDS
END START
```

Трансляция программы и динамической библиотеки из листинга 3.3.4:

```
ml /c /coff dllex2.asm
link /subsystem:windows dllex2.obj
ml /c /coff dll2.asm
link /subsystem:windows  /DLL  /ENTRY:DLLENTRY dll2.obj
```

Далее мы рассмотрим весьма интересный пример (см. листинг 3.3.5). Основной процесс использует ресурсы загруженной им динамической библиотеки. Я уже говорил, что файлы шрифтов, по сути, являются динамическими библиотеками. Не правда ли, удобно: ресурсы можно поместить отдельно от основной программы в динамическую библиотеку, загружая их по мере необходимости? Наша программа вначале загружает пиктограмму из ресурсов динамической библиотеки и устанавливает ее на окно. Если вы будете щелкать правой кнопкой мыши, направив курсор на окно, то будет вызываться процедура из динамической библиотеки, которая будет поочередно устанавливать то один, то другой значок на окно.

Листинг 3.3.5. Пример загрузки ресурса из динамической библиотеки

```
//файл dll3.rc
//идентификаторы
#define IDI_ICON1 3
#define IDI_ICON2 10

//определили пиктограмму
IDI_ICON1 ICON "ico1.ico"
IDI_ICON2 ICON "ico2.ico"

;динамическая библиотека DLL3.ASM
.586P
PUBLIC SETIC
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
WM_SETICON  equ 80h
;прототипы внешних процедур
EXTERN      LoadIconA@8:NEAR
EXTERN      PostMessageA@16:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      PRIZ DB 0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
DLLENTRY PROC
      PUSH EBP
```

```
      MOV  EBP,ESP
      MOV  EAX,1
      LEAVE
      RET 12
DLLENTRY ENDP
; [EBP+8]
; [EBP+0CH]
SETIC PROC EXPORT
      PUSH EBP
      MOV  EBP,ESP
;выбрать, какую пиктограмму устанавливать
      CMP  PRIZ,0
      JZ   IC_1
      MOV  PRIZ,0
      PUSH 3
      JMP  CONT
IC_1:
      MOV  PRIZ,1
      PUSH 10
CONT:
;загрузить пиктограмму из ресурсов библиотеки
      PUSH  DWORD PTR [EBP+0CH] ;идентификатор
;динамической библиотеки
      CALL  LoadIconA@8
;установить значок окна
      PUSH  EAX
      PUSH  0
      PUSH  WM_SETICON
      PUSH  DWORD PTR [EBP+08H] ;дескриптор окна
      CALL  PostMessageA@16
      POP   EBP
      RET   8
SETIC ENDP
_TEXT ENDS
END DLLENTRY

//файл dllex3.rc
//определение констант
#define WS_SYSMENU       0x00080000L
#define WS_MINIMIZEBOX   0x00020000L
#define WS_MAXIMIZEBOX   0x00010000L
#define DS_3DLOOK        0x0004L

//определение диалогового окна
DIAL1 DIALOG 0, 0, 340, 120
```

```
STYLE WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX | DS_3DLOOK
CAPTION "Диалоговое окно с пиктограммой из динамической библиотеки"
FONT 8, "Arial"
{
}

;основной модуль DLLEX3.ASM, вызывающий
;процедуру из динамической библиотеки
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
;сообщение приходит при закрытии окна
WM_CLOSE           equ 10h
WM_INITDIALOG      equ 110h
WM_SETICON         equ 80h
WM_LBUTTONDOWN     equ 201h
;прототипы внешних процедур
EXTERN  PostMessageA@16:NEAR
EXTERN  GetProcAddress@8:NEAR
EXTERN  LoadLibraryA@4:NEAR
EXTERN  FreeLibrary@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  LoadIconA@8:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
      LIBR     DB 'DLL3.DLL',0
      HLIB     DD ?
      HINST    DD ?
      PA       DB "DIAL1",0
      NAMEPROC DB "_SETIC@0",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
```

```
;создать диалог
      MOV  [HINST], EAX
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  [HINST]
CALL  DialogBoxParamA@20
;выход
_EXIT:
      PUSH 0
      CALL ExitProcess@4
;процедура окна
;расположение параметров в стеке
; [BP+014H]  ;LPARAM
; [BP+10H]   ;WAPARAM
; [BP+0CH]   ;MES
; [BP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;----------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
;закрыть библиотеку
;библиотека автоматически закрывается также
;при выходе из программы
      PUSH HLIB
      CALL FreeLibrary@4
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP FINISH
L1:
      CMP DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE L2
;загрузить библиотеку
      PUSH OFFSET LIBR
      CALL LoadLibraryA@4
      MOV  HLIB,EAX
; загрузить пиктограмму
      PUSH  3      ; идентификатор пиктограммы
```

```
      PUSH [HLIB]  ; идентификатор процесса
      CALL LoadIconA@8
;установить пиктограмму
      PUSH EAX
      PUSH 0       ; тип пиктограммы (маленькая)
      PUSH WM_SETICON
      PUSH DWORD PTR [EBP+08H]
      CALL PostMessageA@16
      JMP  FINISH
L2:
      CMP DWORD PTR [EBP+0CH],WM_LBUTTONDOWN
      JNE FINISH
;получить адрес процедуры из динамической библиотеки
      PUSH OFFSET NAMEPROC
      PUSH HLIB
      CALL GetProcAddress@8
;вызвать процедуру с двумя параметрами
      PUSH [HLIB]
      PUSH DWORD PTR [EBP+08H]
      CALL EAX
FINISH:
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      MOV EAX,0
      RET 16
WNDPROC ENDP
_TEXT ENDS
END START
```

Трансляция программ из листинга 3.3.5:

```
ml /c /coff  dllex3.asm
rc dllex3.rc
link /subsystem:windows dllex3.obj dllex3.res
ml /c /coff  dll3.asm
rc dll3.rc
link /subsystem:windows /DLL /ENTRY:DLLENTRY dll3.obj dll3.res
```

Как мы уже не раз с вами убеждались, динамическая библиотека становится частью программы, обладая вместе с процессом всеми ее возможностями. Так, при помощи функций `GetMessage` и `PeekMessage` она может получать сообщения, предназначенные для процесса. Если вы хотите создать в динамической библиотеке окно, то вам следует воспользоваться идентификатором вызвавшей динамическую библиотеку программы.

Совместное использование памяти разными процессами

Рассмотрим теперь вопрос о том, как используют динамическую библиотеку различные экземпляры приложения или разные процессы. Если вы немного знакомы с принципом функционирования операционной системы Windows, то, возможно, такая постановка вопроса у вас вызовет недоумение. "У каждого приложения свое адресное пространство, куда загружается динамическая библиотека", — скажете вы. Конечно, это не совсем рационально, но зато безопасно. О памяти мы еще подробно будем говорить в *главе 3.6*, здесь же заметим, что, вообще говоря, приложение может инициализировать так называемую *разделяемую память*. Мы вернемся к этому вопросу еще неоднократно, сейчас же рассмотрим этот вопрос чисто технически, применительно к динамическим библиотекам. Рассмотрим конкретную ситуацию. Запускаемое приложение загружает динамическую библиотеку и вызывает процедуру из динамической библиотеки, которая меняет данные, расположенные опять же в динамической библиотеке. Запустим теперь второй экземпляр приложения. Оно загружает еще один экземпляр динамической библиотеки. Могут быть ситуации, когда желательно, чтобы второе запущенное приложение "знало", что по команде первого приложения данные уже изменились. Ясно, что в этом случае данные, которыми оперирует динамическая библиотека, должны быть общими. Технически это делается очень просто. У редактора связей LINK.EXE есть опция `/section:` *`имя, атрибуты`*, которая позволяет объявить явно свойства данной секции. Мы будем говорить о секциях далее, здесь же достаточно сказать, секция — это просто сегмент в старом понимании.

Представленные в листинге 3.3.6 программы весьма просты. По сути, они лишь иллюстрируют возможности использования разделяемой памяти. Перед выходом из процедуры динамической библиотеки изменяется строка, хранящаяся в разделяемой секции памяти. При этом приложение не заканчивает своей работы. При запуске второго экземпляра приложения на экран выводится уже измененное первым приложением значение строки.

Листинг 3.3.6. Пример использования разделяемой памяти в динамической библиотеке

```
;динамическая библиотека DLL4.ASM
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
PUBLIC DLLP1
;прототипы внешних процедур
```

```
EXTERN  MessageBoxA@16:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      TEXT DB "В динамической библиотеке",0
      MS   DB "Сообщение",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
; [EBP+10H]   ;резервный параметр
; [EBP+0CH]   ;причина вызова
; [EBP+8]     ;идентификатор DLL-модуля
DLLENTRY:
      PUSH EBP
      MOV  EBP,ESP
      MOV  EAX,1
      LEAVE
      RET  12
;-------------------
;адреса параметров
DLLP1 PROC EXPORT
      PUSH  EBP
      MOV   EBP,ESP
      PUSH  0
      PUSH  OFFSET MS
      PUSH  OFFSET TEXT
      PUSH  0
      CALL  MessageBoxA@16
;изменим строку, расположенную в разделяемой памяти
      MOV   TEXT,'И'
      MOV   TEXT+1,'з'
      POP   EBP
      RET
DLLP1 ENDP
_TEXT ENDS
END DLLENTRY
;основной модуль DLLEX4.ASM, вызывающий
;процедуру из динамической библиотеки
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
;прототипы внешних процедур
```

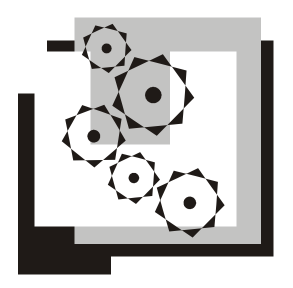
```
EXTERN  GetProcAddress@8:NEAR
EXTERN  LoadLibraryA@4:NEAR
EXTERN  FreeLibrary@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  MessageBoxA@16:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
      TXT  DB 'Ошибка динамической библиотеки',0
      MS   DB 'Сообщение',0
      LIBR DB 'DLL4.DLL',0
      HLIB DD ?
      NAMEPROC DB '_DLLP1@0',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
; [EBP+10H]   ;резервный параметр
; [EBP+0CH]   ;причина вызова
; [EBP+8]     ;идентификатор DLL-модуля
START:
;загрузить библиотеку
      PUSH OFFSET LIBR
      CALL LoadLibraryA@4
      CMP  EAX,0
      JE   _ERR
      MOV  HLIB,EAX
;получить адрес
      PUSH OFFSET NAMEPROC
      PUSH HLIB
      CALL GetProcAddress@8
      CMP  EAX,0
      JNE  YES_NAME
;сообщение об ошибке
_ERR:
      PUSH  0
      PUSH  OFFSET MS
      PUSH  OFFSET TXT
      PUSH  0
      CALL  MessageBoxA@16
      JMP  _EXIT
YES_NAME:
      CALL  EAX
      PUSH  0
```

```
      PUSH  OFFSET MS
      PUSH  OFFSET MS
      PUSH  0
      CALL  MessageBoxA@16
;закрыть библиотеку
;библиотека автоматически закрывается также
;при выходе из программы
      PUSH  OFFSET NAMEPROC
      PUSH  HLIB
      CALL  FreeLibrary@4
;выход
_EXIT:
      PUSH  0
      CALL  ExitProcess@4
_TEXT ENDS
END START
```

Трансляция программ из листинга 3.3.6:

```
ml /c /coff /DMASM dll4.asm
link /subsystem:windows /DLL /section:.data,SRW dll2.obj
ml /c /coff dllex4.asm
link /subsystem:windows dllex4.obj
```

Атрибуты опции `SECTION`: `S` — SHARED (разделяемая), `R` — READ (для чтения), `W` — WRITE (для записи).



Глава 3.4

Сетевое программирование

Данная глава охватывает узкую часть очень обширной области, называемой сетевым программированием. Один вопрос, который нас здесь будет интересовать, — это доступ к ресурсам локальной сети. Другой вопрос сетевого программирования — программирование сокетов — слишком обширен, поэтому я изложу лишь некоторые его основы.

Сетевые устройства

В прикладном программировании часто возникает вопрос определения сетевых устройств. В принципе, вопрос можно поставить более широко: как определить тип того или иного устройства? Тот, кто программировал в MS-DOS, помнит, что там правильное определение типа устройства было непростой задачей. Операционная система Windows облегчает нам задачу. В ОС имеется очень полезная функция `GetDriveType`, единственным аргументом которой является строка корневого каталога искомого устройства, например `"A:\"` или `"D:\"`. По возвращаемому функцией значению мы и определяем тип устройства (см. файл driv.inc в листинге 3.4.1 и табл. 3.4.1). Результат работы программы представлен на рис. 3.4.1.

Таблица 3.4.1. *Значения, возвращаемые функцией* `GetDriveType`

Возвращаемые значения	**Объяснения**
`DRIVE_UNKNOWN = 0`	Устройство не определено
`DRIVE_NO_ROOT_DIR = 1`	Возможно, ошибка монтирования корневого каталога
`DRIVE_REMOVABLE = 2`	Сменный носитель, например, гибкий диск или устройство памяти USB

Таблица 3.4.1 (окончание)

Возвращаемые значения	Объяснения
DRIVE_FIXED = 3	Жесткий диск
DRIVE_REMOTE = 4	Удаленное устройство, например, сетевой диск
DRIVE_CDROM = 5	Накопитель на компакт-диске
DRIVE_RAMDISK = 6	Электронный диск

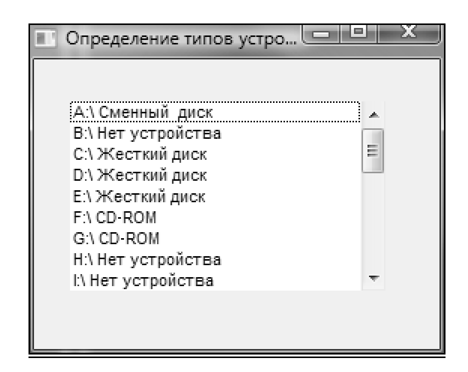


Рис. 3.4.1. Результат работы программы из листинга 3.4.1 на моем домашнем компьютере

Листинг 3.4.1. Простой пример определения типа устройств

```
//файл driv.rc
//определение констант
#define WS_SYSMENU      0x00080000L
#define WS_MINIMIZEBOX  0x00020000L
#define WS_MAXIMIZEBOX  0x00010000L
#define WS_VISIBLE      0x10000000L
#define WS_TABSTOP      0x00010000L
#define WS_VSCROLL      0x00200000L
#define DS_3DLOOK       0x0004L

//идентификаторы
#define LIST1      101

//определение диалогового окна
DIAL1 DIALOG 0, 0, 180, 110
STYLE WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
DS_3DLOOK
CAPTION "Определение типов устройств"
```

```
FONT 8, "Arial"
{
      CONTROL "ListBox1",LIST1,"listbox", WS_VISIBLE |
      WS_TABSTOP | WS_VSCROLL,
      16, 16, 140, 75
}

;файл driv.inc
;константы
;значения, возвращаемые функцией GetDiveType
;значения 0 и 1 можно считать признаком отсутствия устройства
DRIVE_REMOVABLE     equ 2 ;накопитель на гибком диске
DRIVE_FIXED         equ 3 ;устройство жесткого диска
DRIVE_REMOTE        equ 4 ;сетевой диск
DRIVE_CDROM         equ 5 ;накопитель на лазерном диске
DRIVE_RAMDISK       equ 6 ;электронный диск
;сообщение приходит при закрытии окна
WM_CLOSE            equ 10h
WM_INITDIALOG       equ 110h
WM_COMMAND          equ 111h
LB_ADDSTRING        equ 180h
LB_RESETCONTENT     equ 184h
WM_LBUTTONDOWN      equ 201h
;прототипы внешних процедур
EXTERN lstrcpyA@8:NEAR
EXTERN lstrcatA@8:NEAR
EXTERN GetDriveTypeA@4:NEAR
EXTERN ExitProcess@4:NEAR
EXTERN GetModuleHandleA@4:NEAR
EXTERN DialogBoxParamA@20:NEAR
EXTERN EndDialog@8:NEAR
EXTERN SendDlgItemMessageA@20:NEAR
;структура сообщения
MSGSTRUCT STRUC
      MSHWND        DD ?
      MSMESSAGE     DD ?
      MSWPARAM      DD ?
      MSLPARAM      DD ?
      MSTIME        DD ?
      MSPT          DD ?
MSGSTRUCT ENDS

;файл driv.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
```

```
include driv.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      PRIZ   DB 0
      MSG    MSGSTRUCT <?>
      HINST  DD 0 ;дескриптор приложения
      PA     DB "DIAL1",0
      ROO    DB "?:\",0
      BUFER  DB 40 DUP(0)
      TYP0   DB " Нет устройства",0
      TYP1   DB " Нет устройства",0
      TYP2   DB " Сменный  диск",0
      TYP3   DB " Жесткий диск",0
      TYP4   DB " Сетевое устройство",0
      TYP5   DB " CD-ROM",0
      TYP6   DB " Электронный диск",0
      INDEX  DD OFFSET TYP0
             DD OFFSET TYP1
             DD OFFSET TYP2
             DD OFFSET TYP3
             DD OFFSET TYP4
             DD OFFSET TYP5
             DD OFFSET TYP6
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
;-------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  HINST
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
;сообщение об ошибке
KOL:
```

```
;-----------------------------------
      PUSH  0
      CALL  ExitProcess@4
;-----------------------------------
;процедура окна
;расположение параметров в стеке
; [BP+014H]  ;LPARAM
; [BP+10H]   ;WAPARAM
; [BP+0CH]   ;MES
; [BP+8]     ;HWND
WNDPROC PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;----------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP  FINISH
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  L2
L4:
;здесь анализ устройств и заполнение списка
      MOV  ECX,65
LOO:
      PUSH ECX
      MOV  ROO,CL
;определить тип устройства
      PUSH OFFSET ROO
      CALL GetDriveTypeA@4
;полный список
      CMP  PRIZ,0
      JZ  _ALL
      CMP  EAX,2
      JB   L3
_ALL:
;получить индекс
      SHL  EAX,2
      PUSH EAX
;создать строку для списка
      PUSH OFFSET ROO
```

```
      PUSH OFFSET BUFER
      CALL lstrcpyA@8
      POP  EBX
      PUSH INDEX[EBX]
      PUSH OFFSET BUFER
      CALL lstrcatA@8
;отправить строку в список
      PUSH OFFSET BUFER
      PUSH 0
      PUSH LB_ADDSTRING
      PUSH 101
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
L3:
;проверить, не достигнута ли граница цикла
      POP ECX
      INC ECX
      CMP ECX,91
      JNE LOO
      JMP FINISH
L2:
      CMP DWORD PTR [EBP+0CH],WM_LBUTTONDOWN
      JNE FINISH
      PUSH 0
      PUSH 0
      PUSH LB_RESETCONTENT
      PUSH 101
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
      CMP PRIZ,0
      JE  YES_0
      MOV PRIZ,0
      JMP L4
YES_0:
      MOV PRIZ,1
      JMP L4
FINISH:
      MOV EAX,0
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      RET 16
WNDPROC  ENDP
_TEXT ENDS
END START
```

Трансляция программы, представленной в листинге 3.4.1:

```
ml /c /coff  driv.asm
rc driv.rc
link /subsystem:windows driv.obj driv.res
```

После того как вы определили, что данное устройство является сетевым, может возникнуть вопрос, как определить статус устройства? Под статусом в данном случае я понимаю три возможных состояния, в которых может находиться устройство: устройство открыто для чтения и записи, устройство открыто только для чтения, устройство недоступно. Лично я поступаю следующим образом. Для проверки статуса устройства я использую две функции: `CreateFile` и `GetDiskFreeSpace`. С первой функцией вы уже знакомы. С помощью второй функции можно определить объем свободного места на диске. Рассмотрим параметры этой функции:

- 1-й параметр — адрес строки, заканчивающейся нулем, содержащей корневой каталог устройства. Например `"C:\"`. Если параметр равен `NULL`, функция исследует корневой каталог текущего диска;
- 2-й параметр — указатель на переменную типа `DWORD`, которая в результате выполнения функции должна получить количество секторов в одном кластере;
- 3-й параметр — указатель на переменную типа `DWORD`, которая в результате выполнения функции должна получить количество байтов в одном секторе;
- 4-й параметр — указатель на переменную типа `DWORD`, которая в результате выполнения функции должна получить количество свободных кластеров. Если на диске используются квоты, то количество свободных кластеров, получаемых с помощью данной функции, окажется меньше реального количества свободных кластеров;
- 5-й параметр — указатель на переменную типа `DWORD`, которая в результате выполнения функции должна получить количество кластеров на устройстве. Если на диске используются квоты, то количество кластеров, получаемых с помощью данной функции, окажется меньше реального количества кластеров.

Получив информацию от данной функции, вы легко сосчитаете количество байтов на выбранном устройстве и количество свободных байтов.

Если данное устройство позволяет создать файл (файл лучше создавать с атрибутом "удалить после закрытия", тогда операционная система сама выполнит процедуру удаления) и прочесть данные об устройстве (функция `GetDiskFreeSpace`), следовательно, оно открыто для чтения и записи. Если устройство позволяет прочитать данные о нем, но не позволяет создать файл,

следовательно, устройство открыто только для чтения. Наконец, если не разрешено ни то, ни другое, то устройство недоступно. Такой бесхитростный подход работает весьма эффективно.

Поиск сетевых устройств и подключение к ним

В данном разделе мы рассмотрим вопрос о доступе к ресурсам локальной сети. При этом следует выделить две задачи: поиск ресурсов в локальной сети и подключение к ресурсам. Начну с того, что перечислю основные функции для работы с сетевыми ресурсами. Это не все функции, но их вполне достаточно, чтобы ваша программа самостоятельно искала сетевые ресурсы и подключалась к ним. Конечно, я предполагаю, что вы умеете работать в сети, знаете, что такое сетевое устройство, сетевой компьютер и т. п.

Прежде всего, рассмотрим структуру, которая используется в данных функциях:

```
NETRESOURCE STRUC
      dwScope         DWORD ?
      dwType          DWORD ?
      dwDisplayType   DWORD ?
      dwUsage         DWORD ?
      lpLocalName     DWORD ?
      lpRemoteName    DWORD ?
      lpComment       DWORD ?
      lpProvider      DWORD ?
NETRESOURCE ENDS
```

Здесь:

- `DwScope` — может принимать одно из трех значений:
  - `RESOURCE_CONNECTED` — ресурс подсоединен в настоящее время;
  - `RESOURCE_REMEMBERED` — ресурс, запоминаемый системой, чтобы при запуске автоматически подсоединяться к нему;
  - `RESOURCE_GLOBALNET` — глобальный сетевой ресурс. Скорее всего, вам понадобится только последнее значение;
- `DwType` — тип ресурса. Возможны следующие значения:
  - `RESOURCETYPE_ANY` — любой ресурс;
  - `RESOURCETYPE_DISK` — диск;
  - `RESOURCETYPE_PRINT` — сетевой принтер;

- DwDisplayType — как данный ресурс должен быть представлен сетевым браузером. В настоящее время типов всего четыре:
  - RESOURCEDISPLAYTYPE_SERVER — сетевой объект может рассматриваться как сервер;
  - RESOURCEDISPLAYTYPE_DOMAIN — ресурс рассматривается как домен;
  - RESOURCEDISPLAYTYPE_GENERIC — тип данного ресурса не имеет значения;
  - RESOURCEDISPLAYTYPE_SHARE — устройство общего доступа;
- dwUsage — чаще всего полагают равным 0. Данный параметр берется во внимание, если только параметр dwScope равен RESOURCE_GLOBALNET. В этом случае он может принять одно из двух значений: RESOURCEUSAGE_CONNECTABLE=1h (к ресурсу можно подсоединиться) и RESOURCEUSAGE_CONTAINER=2h (ресурс представляет собой контейнер);
- lpLocalName — локальное имя устройства, например E:, LPT1: и т. п.;
- lpRemoteName — сетевое имя, например \\SUPER, \\NDI\EPSON и т. д.;
- lpComment — комментарий к сетевому ресурсу;
- lpProvider — имя провайдера. Например, в качестве провайдера могут быть Microsoft Network или NetWare. Если значение провайдера неизвестно, тогда значение параметра равно NULL.

Рассмотрим сетевые функции.

WNetAddConnection2 — с помощью данной функции можно подсоединиться к сетевому ресурсу (диску или принтеру).

- 1-й параметр — адрес структуры NETRESOURCE, значение полей которой было разобрано выше. Должны быть заполнены следующие поля: dwType, lpLocalName, lpRemoteName, lpProvider (обычно NULL). Ниже будет приведен пример заполнения.
- 2-й параметр — пароль, необходимый для соединения с ресурсом. В случае пустой сроки — соединение беспарольное, в случае NULL — берется пароль, ассоциированный с именем (см. ниже).
- 3-й параметр — имя пользователя. Если значение NULL, то берется имя по умолчанию.
- 4-й параметр определяет, будет ли система потом автоматически подсоединяться к данному ресурсу. В случае значения 0 такое подсоединение не происходит[1].

[1] Точнее, если нулю равен нулевой бит параметра. Но на этих подробностях мы останавливаться не будем.

При успешном завершении функция возвращает 0 (NO_ERROR). Это касается и всех остальных рассматриваемых ниже функций.

WNetCancelConnection2 — отсоединить сетевой ресурс.

- 1-й параметр содержит указатель на строку с именем ресурса. Причем если имя локальное, то разрывается данное локальное соединение. Если это имя удаленного ресурса, то разрываются все соединения с данным ресурсом.
- 2-й параметр определяет, будет ли система и далее подсоединяться к данному ресурсу. Если 0, то подсоединение (если было) будет возобновляться при следующем запуске системы.
- 3-й параметр — если значение не нулевое, то отсоединение произойдет, даже если на сетевом диске имеются открытые файлы или сетевой принтер выполняет задание с данного компьютера.

WnetOpenEnum — открыть поиск сетевых ресурсов. Вообще говоря, сетевые ресурсы образуют древообразную структуру. Об этом мы будем говорить позже, поэтому поиск сетевых ресурсов весьма напоминает поиск файлов по дереву каталогов.

- 1-й параметр — dwScope (см. структуру NETRESOURCE), обычно полагают RESOURCE_GLOBALNET.
- 2-й параметр — dwType, для поиска всяких ресурсов следует положить равным RESOURCETYPE_ANY.
- 3-й параметр — dwUsage, обычно следует положить равным нулю.
- 4-й параметр — адрес структур NETRESOURCE. Если адрес равен 0 (NULL), то поиск будет начинаться с самого нижнего уровня (корня), в противном случае поиск начнется с уровня, определяемого полями lpRemoteName и lpProvider.
- 5-й параметр — указатель на переменную, которая должна получить дескриптор для дальнейшего поиска.

WnetCloseEnum — закрыть поиск. Единственным параметром этой функции является дескриптор, полученный при выполнении функции WnetOpenEnum.

WnetEnumResource — функция, осуществляющая непосредственный поиск сетевых ресурсов.

- 1-й параметр — дескриптор поиска.
- 2-й параметр — указатель на переменную, содержащую значение максимального количества ресурсов, которое должно быть найдено за один раз. Обычно переменную полагают равной 0FFFFFFFFH — для поиска всех возможных ресурсов. При успешном завершении данной функции переменная будет содержать количество реально найденных ресурсов.

- 3-й параметр — указатель на массив, каждым элементом которого является структура NETRESOURCE. Ясно, что данный массив должен быть достаточно большим, чтобы вместить столько данных о ресурсе, сколько вам нужно. Поскольку размер структуры составляет 32 байта, то в случае большой сети размер массива должен составлять не менее 32 000 байтов.
- 4-й параметр — адрес переменной, содержащей объем массива. Если объем окажется мал, то переменная будет содержать реально требуемый объем.

Лишний раз подчеркну, что данная функция осуществляет поиск не всех ресурсов, а лишь ресурсов данного иерархического уровня. Так что без рекурсии не обойтись.

WNetGetConnection — с помощью этой функции можно получить информацию о данном соединении.

- 1-й параметр — адрес буфера, куда должно быть помещено локальное имя устройства (A:, C:, LPT2 и т. п.).
- 2-й параметр — адрес буфера, куда будет помещено удаленное имя устройства.
- 3-й параметр — указатель на переменную, содержащую размер буфера.

Завершая краткий обзор сетевых функций и переходя к программированию, замечу, что нами описана лишь часть наиболее важных сетевых функций. Полную информацию о сетевых функциях для доступа к ресурсам локальной сети можно получить в документальной базе данных MSDN.

В листинге 3.4.2 представлена программа, позволяющая подключаться к сетевым дискам, разрешенным к доступу. Командная строка программы: PROG \\SERVER\CC Z:. Первый параметр — имя подключаемого устройства, включающее имя сетевого сервера. Второй параметр — локальный диск, на который будет спроецирован сетевой диск.

Листинг 3.4.2. Программа, осуществляющая соединение с сетевым ресурсом

```
;программа, осуществляющая подсоединение к
;сетевому ресурсу, например: prog \\SUPER\\D Z:
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
RESOURCETYPE_ANY equ 0
;прототипы внешних процедур
EXTERN  CharToOemA@8:NEAR
EXTERN  lstrcatA@8:NEAR
```

```
EXTERN  lstrlenA@4:NEAR
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetCommandLineA@0:NEAR
EXTERN  WNetAddConnection2A@16:NEAR
;структуры
NETRESOURCE STRUC
      dwScope        DWORD ?
      dwType         DWORD ?
      dwDisplayType  DWORD ?
      dwUsage        DWORD ?
      lpLocalName    DWORD ?
      lpRemoteName   DWORD ?
      lpComment      DWORD ?
      lpProvider     DWORD ?
NETRESOURCE ENDS
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\mpr.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      BUF1  DB  100 dup(0)
      BUF2  DB  100 dup(0)
      LENS  DWORD ? ;количество выведенных символов
      HANDL DWORD ?
      NR    NETRESOURCE <0>
      ERR2  DB "Ошибка!",0
      ERR1  DB "Мало параметров!",0
      ST1   DB "->",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор выхода вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;получить количество параметров
      CALL NUMPAR
      CMP  EAX,3
      JNB  PAR_OK
      LEA  EBX,ERR1
      CALL SETMSG
```

```
      JMP _END
PAR_OK:
;получить параметры
      MOV  EDI,2
      LEA  EBX,BUF1
      CALL GETPAR
      MOV  EDI,3
      LEA  EBX,BUF2
      CALL GETPAR
;пытаемся произвести подсоединение
;вначале заполняем структуру NETRESOURCE
      MOV NR.dwType,RESOURCETYPE_ANY
      LEA EAX,BUF2
      MOV NR.lpLocalName,EAX
      LEA EAX,BUF1
      MOV NR.lpRemoteName,EAX
      MOV NR.lpProvider,0
;вызов функции, осуществляющей соединение
      PUSH  0
      PUSH  0   ; имя по умолчанию
      PUSH  0   ; пароль, ассоциированный с именем
      PUSH  OFFSET NR
      CALL  WNetAddConnection2A@16
      CMP   EAX,0
      JE    _OK
;сообщение об ошибке
      LEA   EBX,ERR2
      CALL  SETMSG
      JMP   _END
_OK:
;сообщение об успешном соединении
      PUSH OFFSET ST1
      PUSH OFFSET BUF1
      CALL lstrcatA@8
      PUSH OFFSET BUF2
      PUSH OFFSET BUF1
      CALL lstrcatA@8
      LEA  EBX,BUF1
      CALL SETMSG
_END:
      PUSH 0
      CALL ExitProcess@4
;определить количество параметров (->EAX)
NUMPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX ;указатель на строку
```

```
      XOR  ECX,ECX ;счетчик
      MOV  EDX,1 ;признак
L1:
      CMP  BYTE PTR [ESI],0
      JE   L4
      CMP  BYTE PTR [ESI],32
      JE   L3
      ADD  ECX,EDX ;номер параметра
      MOV  EDX,0
      JMP  L2
L3:
      OR  EDX,1
L2:
      INC ESI
      JMP L1
L4:
      MOV EAX,ECX
      RET
NUMPAR ENDP
;получить параметр
;EBX - указывает на буфер, куда будет помещен параметр
;в буфер помещается строка с нулем на конце
;EDI - номер параметра
GETPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX   ; указатель на строку
      XOR  ECX,ECX   ; счетчик
      MOV  EDX,1     ; признак
L1:
      CMP  BYTE PTR [ESI],0
      JE   L4
      CMP  BYTE PTR [ESI],32
      JE   L3
      ADD  ECX,EDX   ; номер параметра
      MOV  EDX,0
      JMP  L2
L3:
      OR   EDX,1
L2:
      CMP  ECX,EDI
      JNE  L5
      MOV  AL,BYTE PTR [ESI]
      CMP  AL,32
      JE   L5
      MOV  BYTE PTR [EBX],AL
      INC  EBX
```

```
L5:
      INC  ESI
      JMP  L1
L4:
      MOV  BYTE PTR [EBX],0
      RET
GETPAR ENDP
;вывод сообщения
;EBX -> строка
SETMSG PROC
      PUSH EBX
      PUSH EBX
      CALL CharToOemA@8
      PUSH EBX
      CALL lstrlenA@4
      PUSH 0
      PUSH OFFSET LENS
      PUSH EAX
      PUSH EBX
      PUSH HANDL
      CALL WriteConsoleA@20
      RET
SETMSG ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.4.2:

```
ml /c /coff net.asm
link /subsystem:console net.obj
```

А теперь комментарий к программе из листинга 3.4.2.

- Рассматривая программу из листинга 3.4.2, прежде всего, обратите внимание на использование локальных меток. MASM32 распознает локальные метки автоматически, но так поступают не все ассемблеры.
- Отмечу одну существенную особенность функции `WNetAddConnection2A@16`. При заполнении структуры `NETRESOURCE` поле `dwType` для операционной системы семейства Windows NT должно заполняться нулем, что соответствует константе `RESOURCETYPE_ANY`. В операционных системах семейства Windows 9*x* были некоторые отличия, но я больше не рассматриваю эти системы в своей книге.
- Данная программа далека от совершенства. Попробуйте над ней поработать. В частности, не мешало бы проверять, является ли локальное устройство сетевым или нет, и если является, спросить разрешение на повторное подключение данного устройства. В этом случае необходимо

вначале отключить устройство от старого сетевого ресурса при помощи известной вам функции `WNetCancelConnection2`.

- Если необходимо подключать сетевой принтер, то поле `dwType` должно быть равно `RESOURCETYPE_PRINT`.

Следующая программа (листинг 3.4.3) осуществляет рекурсивный поиск ресурсов локальной сети. Работая в консольном режиме, она выдает на экран название провайдера и удаленное имя ресурса. Данная программа должна правильно работать и в сетях Microsoft, и в сетях Novel (результат работы программы можно увидеть в листинге 3.4.4).

Листинг 3.4.3. Рекурсивный поиск ресурсов в локальной сети

```
;программа NET1, осуществляющая поиск сетевых ресурсов
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE  equ -11
RESOURCETYPE_DISK  equ 1h
RESOURCE_GLOBALNET equ 2h
RESOURCETYPE_ANY   equ 0h
;прототипы внешних процедур
EXTERN  CharToOemA@8:NEAR
EXTERN  RtlMoveMemory@12:NEAR
EXTERN  WNetCloseEnum@4:NEAR
EXTERN  WNetEnumResourceA@16:NEAR
EXTERN  WNetOpenEnumA@20:NEAR
EXTERN  lstrcpyA@8:NEAR
EXTERN  lstrcatA@8:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetCommandLineA@0:NEAR
;структуры
NETRESOURCE STRUC
      dwScope        DWORD ?
      dwType         DWORD ?
      dwDisplayType  DWORD ?
      dwUsage        DWORD ?
      lpLocalName    DWORD ?
      lpRemoteName   DWORD ?
      lpComment      DWORD ?
      lpProvider     DWORD ?
```

```
NETRESOURCE ENDS

;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\mpr.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
      LENS  DWORD ? ;количество выведенных символов
      HANDL DWORD ?
      NR    NETRESOURCE <0>
      ENT   DB 13,10,0
      BUF   DB 100 dup(0)
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор выхода вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;запустить процедуру поиска сетевых ресурсов
      PUSH 0
      PUSH OFFSET NR
      CALL POISK
_END:
      PUSH 0
      CALL ExitProcess@4
;процедура поиска
POISK PROC
      PAR1 EQU [EBP+8]     ; указатель на структуру
      PAR2 EQU [EBP+0CH]   ; признак
;локальные переменные
      HANDLP EQU [EBP-4]   ; дескриптор поиска
      CC     EQU [EBP-8]
      NB     EQU [EBP-12]
      NR1    EQU [EBP-44] ; структура
      BUFER  EQU [EBP-144]; буфер
      RS     EQU [EBP-32144] ; массив структур
      PUSH   EBP
      MOV    EBP,ESP
      SUB    ESP,32144
      CMP    DWORD PTR PAR2,0
      JNE    SECOND
;при первом запуске NULL
      XOR    EBX,EBX
```

```
      JMP    FIRST
SECOND:
;запуск при рекурсивном вызове
;вначале скопировать структуру в локальную переменную,
;хотя для данной программы это излишне
      PUSH   32
      PUSH   DWORD PTR PAR1
      LEA    EAX,DWORD PTR NR1
      PUSH   EAX
      CALL   RtlMoveMemory@12
;при вторичном поиске указатель на структуру
      LEA    EBX,DWORD PTR NR1
FIRST:
;запуск при первом вызове
      LEA    EAX,HANDLP
      PUSH   EAX
      PUSH   EBX
      PUSH   0
      PUSH   RESOURCETYPE_ANY
      PUSH   RESOURCE_GLOBALNET
      CALL   WNetOpenEnumA@20
      CMP    EAX,0
      JNE    _EN
;здесь осуществляется основной поиск
REPI:
;запуск функции WNetEnumResource
;объем массива структур NETRESOURCE
      MOV  DWORD PTR NB,32000
      LEA  EAX,NB
      PUSH EAX
      LEA  EAX,RS
      PUSH EAX
;искать максимальное количество объектов
      MOV  DWORD PTR CC,0FFFFFFFFH
      LEA  EAX,CC
      PUSH EAX
      PUSH DWORD PTR HANDLP
      CALL WNetEnumResourceA@16
      CMP  EAX,0
      JNE _CLOSE
;цикл по полученному массиву
      MOV  ESI,CC
      SHL  ESI,5 ;умножаем на 32
      MOV  EDI,0
LOO:
      CMP EDI,ESI
```

```
      JE  REPI
;вывод информации
;провайдер
      MOV  EBX,DWORD PTR RS[EDI]+28
      CALL SETMSG
;удаленное имя
      MOV  EBX,DWORD PTR RS[EDI]+20
      CALL SETMSG
;сохранить нужные регистры
      PUSH  ESI
      PUSH  EDI
;теперь рекурсивный вызов
      PUSH 1
      LEA  EAX,DWORD PTR RS[EDI]
      PUSH EAX
      CALL POISK
;восстановить регистры
      POP  EDI
      POP  ESI
      ADD  EDI,32
      JMP  LOO
;-----------------------------------
      JMP  REPI
;-----------------------------------
_CLOSE:
      PUSH DWORD PTR HANDLP
      CALL WNetCloseEnum@4
_EN:
      MOV ESP,EBP
      POP EBP
      RET 8
POISK ENDP
;вывод сообщения
;EBX -> строка
SETMSG PROC
;скопировать текст в отдельный буфер
      PUSH EBX
      PUSH OFFSET BUF
      CALL lstrcpyA@8
      LEA  EBX,BUF
;перекодировать для консоли
      PUSH EBX
      PUSH EBX
      CALL CharToOemA@8
;добавить перевод строки
      PUSH OFFSET ENT
```

```
      PUSH EBX
      CALL lstrcatA@8
;определить длину строки
      PUSH EBX
      CALL lstrlenA@4
;вывести строку
      PUSH 0
      PUSH OFFSET LENS
      PUSH EAX
      PUSH EBX
      PUSH HANDL
      CALL WriteConsoleA@20
      RET
SETMSG ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.4.3:

```
ml /c /coff net1.asm
link /subsystem:console /STACK:1000000,1000000 net1.obj
```

Программа в листинге 3.4.3 довольно сложна и требует серьезных пояснений. Прежде всего, хочу сказать, что, если читатель действительно хочет разобраться в сетевом программировании (я в данном случае имею в виду локальную сеть), то необходимо самостоятельно написать несколько программ. Мои программы должны служить отправными точками.

- Нам уже приходилось сталкиваться с локальными переменными, когда мы рассматривали поиск файлов по дереву каталогов. Эта задача весьма похожа, но есть и отличие. В данном случае мы используем слишком большой объем для локальных переменных. По этой причине мы явно указываем (заказываем) большой объем стека (опции `STACK`). По умолчанию компоновщик устанавливает всего 8 Кбайт, что явно недостаточно.
- Функция `WnetEnumResource` требует указать своим параметром массив структур `NETRESOURCE`. Объем одной структуры — 32 байта. Мы резервируем тысячу таких структур. "Не много ли?" — спросите вы. Честно говоря, я не встречал локальной сети с тысячью сетевых компьютеров. Однако я встречал локальную сеть, где на одном из серверов было создано около восьмисот сетевых каталогов. Если говорить на чистоту, то здесь я все же демонстрирую не лучший стиль программирования. Более корректный путь заключается в том, что функция `WnetEnumResource` вначале вызывается с указанием объема буфера меньше, чем 32 байта, — в этом случае в переменную, содержащую объем буфера, будет возвращен необходимый объем. Зная необходимый объем, программа должна запросить у системы

нужный и вторично запустить `WnetEnumResource`. Данный подход более корректен, но и более сложен. Попробуйте реализовать его самостоятельно.

- При рекурсивном вызове процедуры `POISK` первым параметром является указатель на элемент массива структур `NETRESOURCE`. Мы копируем элемент массива в локальную переменную `NR1`. В принципе, в данной программе можно этого не делать, а сразу воспользоваться полученным указателем. В более сложных программах, однако, скорее всего, придется это делать.
- Обратите внимание, что в процедуре вывода информации мы копируем строку в буфер, который потом используем для вывода. Это не прихоть, а необходимость. Дело в том, что пред выводом мы добавляем в конец строки коды 13 и 10. Поскольку мы выводим строки, которые потом используем для дальнейшего поиска, нам приходится использовать для вывода дополнительный буфер.

В листинге 3.4.4 можно видеть пример работы программы из листинга 3.4.3 в одной из локальных компьютерных сетей.

Листинг 3.4.4. Фрагмент листинга — результата работы программы из листинга 3.4.3

```
Службы терминалов Microsoft
Службы терминалов Microsoft
Microsoft Windows Network
Microsoft Windows Network
Microsoft Windows Network
ALTDOMAIN
Microsoft Windows Network
\\JEKA
Microsoft Windows Network
\\PST
Microsoft Windows Network
\\VCENTER
Microsoft Windows Network
SHGPI
Microsoft Windows Network
\\PYTHON
Microsoft Windows Network
\\PYTHON\softinfa
Microsoft Windows Network
\\PYTHON\video
Microsoft Windows Network
\\PYTHON\uploads
Microsoft Windows Network
\\PYTHON\music
```

```
Microsoft Windows Network
WORKGROUP
Microsoft Windows Network
\\ADMNET
Microsoft Windows Network
\\DMS_W
Microsoft Windows Network
\\COMP2_1\AppServ
Microsoft Windows Network
\\COMP2_1\www
Microsoft Windows Network
\\COMP2_10
Microsoft Windows Network
\\COMP2_10\123
Microsoft Windows Network
\\COMP2_6
Microsoft Windows Network
\\COMP2_8
Microsoft Windows Network
\\COMP2_9
Microsoft Windows Network
\\COMP3_1
Microsoft Windows Network
\\COMP3_10
Microsoft Windows Network
\\COMP3_11
Microsoft Windows Network
\\COMP3_12
Microsoft Windows Network
\\COMP3_12\123
Microsoft Windows Network
\\COMP3_12\music
Microsoft Windows Network
\\NITOW\845
Microsoft Windows Network
\\NITOW\www
Microsoft Windows Network
\\NITOW\usr
Microsoft Windows Network
\\OLYA
Microsoft Windows Network
\\OLYA\HP_1200_OLYA
Microsoft Windows Network
\\PIROGOV
Microsoft Windows Network
\\SIN1
```

```
Microsoft Windows Network
\\SQLPI
Microsoft Windows Network
\\SQLPI\Multimedia
Microsoft Windows Network
\\SQLPI\wwwroot
Web Client Network
Web Client Network
```

О сетевых протоколах TCP/IP

Материал данного раздела несколько не вписывается в книгу, но далее рассматривается тема о программировании сокетов, и я вынужден дать элементарный обзор вопросов, с которыми вы обязательно встретитесь, если и далее будете интересоваться сокетами.

О модели OSI

В модели OSI средства сетевого взаимодействия разбиваются на семь уровней. OSI расшифровывается как Open Systems Interconnection и разработана международной организацией по стандартам. Уровни модели рассмотрены в табл. 3.4.2.

Таблица 3.4.2. *Уровни модели OSI*

Название уровня	**Функции уровня**
Физический уровень	Выполняет передачу битов по физическим каналам, таким, как коаксиальный кабель, витая пара или оптоволоконный кабель. На этом уровне определяются характеристики физических сред передачи данных и параметров электрических сигналов. Этот уровень называется еще аппаратным
Канальный уровень	Обеспечивает передачу кадра данных между любыми узлами в сетях с типовой топологией либо между двумя соседними узлами в сетях с произвольной топологией. В протоколах канального уровня заложена определенная структура связей между компьютерами и способы их адресации. Адреса, используемые на канальном уровне в локальных сетях, часто называют MAC-адресами (MAC означает Medium Access Control — управление доступом к среде). Канальный уровень делят еще на два подуровня: • MAC-уровень; • уровень управления логическим каналом — Logical Link Control (LLC)

Таблица 3.4.2 (окончание)

Название уровня	Функции уровня
Сетевой уровень	Обеспечивает доставку данных между любыми двумя узлами в сети с произвольной топологией, при этом он не берет на себя никаких обязательств по надежности передачи данных
Транспортный уровень	Обеспечивает передачу данных между любыми узлами сети с требуемым уровнем надежности. Для этого на транспортном уровне имеются средства установления соединения, нумерации, буферизации и упорядочивания пакетов и учет дублирующих пакетов
Сеансовый уровень	Предоставляет средства управления диалогом, позволяющие фиксировать, какая из взаимодействующих сторон является активной в настоящий момент, а также предоставляет средства синхронизации в рамках процедуры обмена сообщениями. Этот уровень называется еще сессионным
Уровень представления	Уровень представления (Presentation) имеет дело с внешним представлением данных. На этом уровне могут выполняться различные виды преобразования данных, такие как компрессия и декомпрессия, шифровка и дешифровка данных
Прикладной уровень	Прикладной уровень (Application) — это набор разнообразных сетевых сервисов, предоставляемых конечным пользователям и приложениям. Примерами таких сервисов являются, например, электронная почта, передача файлов, подключение удаленных терминалов к компьютеру по сети

Подробнее о модели OSI вы можете прочесть в старой, но замечательной книге Барри Нанса [18].

О семействе TCP/IP

Протоколы семейства TCP/IP (Transmission Control Protocol/Internet Protocol) образуют четырехуровневую структуру. Эта структура схематично изображена на рис. 3.4.2, где в частности показана проекция этих протоколов на модель OSI.

Семейство протоколов TCP/IP имеет давнюю историю, и, тем не менее, эти протоколы являются наиболее используемыми в настоящее время. Нет ни одной современной операционной системы, которая не поддерживала бы этих протоколов. Отчасти это объясняется тем, что на основе этих протоколов построена глобальная компьютерная сеть Интернет.

Прикладной	HTTP, FTP, SMTP, SNMP, telnet
Транспортный	TCP, UDP
Сетевой	IP, RIP, ARP, ICMP, OSPF
Физический	Драйверы устройств

Рис. 3.4.2. Семейство сетевых протоколов

Самый нижний уровень в протоколах TCP/IP (он считается четвертым) не регламентируется, но поддерживает все известные протоколы физического и канального уровня.

Следующий, третий уровень является уровнем межсетевого взаимодействия. Он занимается передачей пакетов с использованием транспортных технологий локальный сетей, транспортных сетей, линий связи и т. п. В качестве основного протокола сетевого уровня используется протокол IP. Протокол IP является дейтаграммным *(см. далее)*, он не гарантирует доставку пакета к месту назначения. К этому уровню относятся все протоколы, связанные с составлением и модификацией таблиц маршрутизации. Это протоколы RIP (Routing Internet Protocol), OSPF (Open Shortest Path First) для сбора маршрутной информации, ICMP (Internet Control Message Protocol) — протокол межсетевых управляющих сообщений.

Замечание

Под дейтаграммой понимается сетевой пакет, который передается независимо от других пакетов и без установки логического соединения.

Второй уровень считается основным. На этом уровне работают протоколы TCP и UDP[2] (User Datagram Protocol, протокол дейтаграмм пользователя). Протокол TCP обеспечивает передачу сообщений между удаленными процессами, образуя виртуальные соединения. Протокол UDP обеспечивает передачу прикладных пакетов дейтаграммным способом (подобно IP) и выполняет функции связующего звена между IP и прикладными процессами.

Первый уровень называется прикладным. Это протоколы высокого уровня. Например, протокол HTTP позволяет передавать информацию в виде Web-

[2] Когда таким образом говорят о протоколах TCP/IP, то имеют в виду, по крайней мере, два протокола или даже целое семейство.

страниц. Протокол FTP позволяет обмениваться файлами между узлами глобальной сети.

Об IP-адресации

Главное в компьютерной сети — это возможность быстро находить нужного адресата. В IP-сетях можно выделить три уровня адресации.

- Физическая или локальная адресация. В локальной сети она основывается на номере сетевого адаптера. Эти адреса уникальны, назначаются производителями оборудования и состоят из шести байтов. В глобальной сети локальные адреса назначаются администратором.
- IP-адрес состоит из четырех байтов. Принято записывать IP-адреса в десятичном виде, отделяя точкой байты: 137.50.50.83. Адрес назначается администратором или выделяется системой автоматически. Адрес состоит из адреса сети и адреса узла *(см. далее)*. Узел может входить в несколько сетей и поэтому иметь несколько IP-адресов. IP-адрес не зависит от физического адреса, поэтому является характеристикой не компьютера, а именно логического узла.
- Символьный адрес или идентификатор может состоять из нескольких частей, отделенных точкой, и назначается администратором. Например, имя **Serv1.bank.COM** состоит из трех частей: **COM** — имя домена, **bank** — название организации, **Serv1** — название компьютера. В локальной сети организации, где я работаю, всего один домен Aogas и символьные имена компьютеров имеют следующий вид: VLAD.Aogas, Igor.Aogas и т. д.

Обратимся снова к IP-адресам. Записанный нами ранее адрес 137.50.50.83 может быть представлен и в двоичном виде

```
10001001 00110010 00110010 01010011
```

На рис. 3.4.3 представлено пять классов IP-адресов. Вы видите, что только три первых класса A, B, C адресуют компьютеры (узлы). Выбор класса адресов обуславливается тем, с какой сетью мы имеем дело (большая, маленькая, средняя).

- Сети класса A имеют номер в диапазоне 1—126. Ноль не используется, а 127 зарезервирован. В сетях этого класса число узлов может быть весьма большим.
- Сети класса B — сети средних размеров. Под номер адреса узла отводятся два байта. Стало быть, адрес 137.50.50.83 моего компьютера говорит нам, что сеть, в которой он работает, относится к типу B.
- Сеть класса C — это сеть малых размеров.

- Адрес класса D — групповой адрес. Если посылаемый пакет имеет пунктом назначения этот адрес, то его получат все компьютеры, имеющие этот адрес. Пакеты с таким адресом называют мультивещательными.
- Класс адресов E — зарезервированная группа адресов.



Рис. 3.4.3. Классы адресов IP

В протоколе IP существует несколько специальных адресов. Вот они:

- адрес, состоящий из нулей. Обозначает адрес того узла, который сгенерировал этот пакет;
- в адресе все нули, кроме адреса узла. По умолчанию предполагается, что адрес относится к той же сети, что и отправитель;
- адрес 255.255.255.255 — так называемый широковещательный адрес. Пакет рассылается всем узлам текущей сети;
- если вместо адреса узла стоят числа 255, т. е. единицы в двоичном представлении;
- адрес 127.0.0.1 применяется для организации обратной связи и называется loopback (что можно перевести как "обратная петля").

Маскирование адресов

Вероятно, вы обратили внимание, что при задании IP-адреса задается еще и маска. Маска — это число, двоичные разряды которой, равные единице, указывают, что именно эти разряды в адресе должны интерпретироваться как

адрес сети. Маскирование, т. е. наложение маски на адреса, позволяет разбивать сеть на несколько подсетей. Что бывает необходимо, если в сети имеются компьютеры, слабо взаимодействующие друг с другом.

Физические адреса и адреса IP

В отличие от такого сетевого протокола как IPX, адреса в протоколе IP не привязываются к компьютеру. Адреса IP на самом деле используются для передачи информации между сетями, а в пределах одной локальной сети пакеты передаются по локальному адресу. Следовательно, должен быть некоторый механизм трансляции IP-адреса в локальный и обратно. Для определения локального адреса по IP-адресу используется протокол ARP (Address Resolution Protocol). Протокол ARP может работать по-разному, в зависимости от того, что собой представляет локальная адресация в этой сети. Существует протокол, который решает обратную задачу. Это протокол RARP.

Узел, которому необходимо выполнить отображение IP-адреса на локальный адрес, формирует ARP-запрос (для разных сетей структура этого запроса может быть различной). Этот запрос рассылается широковещательно в пределах одной сети. Все узлы получают этот запрос и сравнивают свой IP-адрес с адресом в запросе. В случае совпадения узел формирует ответ, где указывает свой локальный адрес и свой IP-адрес.

В локальной сети процесс получения локального адреса происходит автоматически. В глобальных сетях используются специальные таблицы соответствия. Эти таблицы могут храниться на специальном маршрутизаторе, так что запрос отправляется именно ему.

О службе DNS

Служба DNS (Domain Name System, система доменных имен) обеспечивает автоматизацию отображения IP-адресов на символьные адреса. DNS представляет собой распределенную базу данных. Протокол DNS является протоколом прикладного уровня. Этот протокол оперирует такими понятиями, как DNS-клиенты и DNS-серверы. Все DNS-серверы выстроены в логическую иерархическую структуру. Клиент опрашивает эти серверы, пока не обнаружит нужную информацию (соответствие).

В Интернете домены верхнего уровня соответствуют странам, а также назначаются на организационной основе (например, **com** означает, что владельцем сервера является коммерческая организация и т. д.).

Автоматическое назначение IP-адресов

Ручное назначение IP-адресов узлу довольно утомительное дело. Обычно рекомендуется уходить от ручного назначения для сетей с количеством компьютеров 50 и более. Для автоматизации назначения IP-адресов был разработан протокол DHCP (Dynamic Host Configuration Protocol, динамический протокол конфигурации сервера). Протокол позволяет не только полностью автоматизировать процесс назначения адреса, но дать возможность выполнять полуавтоматическую настройку администратором. Следует различать автоматическое и динамическое назначение адреса. При автоматическом назначении раз назначенный адрес будет каждый раз выдаваться компьютеру с данным локальным адресом. При динамическом назначении адрес может выдаваться на некоторое время. При динамическом распределении адресов количество используемых адресов может быть много меньше количества компьютеров в сети.

Маршрутизация и ее принципы

В процессе маршрутизации, т. е. процессе прохождение пакета от начального пункта к конечному, участвуют как маршрутизаторы, так и отдельные узлы. Специальные таблицы маршрутизации могут быть не только у маршрутизатора, но и у обычного узла — компьютера.

В строке таблицы маршрутизации содержатся, по крайней мере, четыре поля: адрес сети назначения, адрес следующего маршрутизатора, номер выходного порта, расстояние до сети назначения. Последняя величина может интерпретироваться самыми разными способами. Например, это может быть какая-то временна́я характеристика или количество узлов, которые должен пройти пакет. Если в таблице маршрутизации имеется не одна строка с одним и тем же адресом строки назначения, то, как правило, выбирается строка с наименьшим значением этого поля.

Использование таких таблиц предполагает так называемую одношаговую маршрутизацию. Возможна и многошаговая маршрутизация, когда посылаемый пакет уже имеет информацию обо всех маршрутизаторах, которые он должен пройти. Такая схема применяется в основном в отладочных ситуациях.

Для отправки пакета следующему маршрутизатору используется вначале протокол ARP, т. к. таблица маршрутизации не содержит в себе локальный адрес.

Если узел или маршрутизатор "видят", что данный адрес относится к локальной сети, то принимается решение о передаче пакета конкретному узлу, с использованием протокола ARP для определения локального адреса.

Таблица маршрутизации содержит также строки, где указаны адреса сетей, непосредственно подключенных к данному маршрутизатору с нулями в поле, которое содержит расстояние до сетей.

Таблица маршрутизации, как правило, имеет строку default (по умолчанию), которая содержит адрес маршрутизатора "по умолчанию". Если в таблице не найдена строка с нужным адресом, то пакет передается этому маршрутизатору. Предполагается, что таким образом пакет дойдет до так называемых магистральных маршрутизаторов, которые содержат исчерпывающие таблицы маршрутизации.

При составлении таблиц используются три типа алгоритмов маршрутизации.

- Фиксированная маршрутизация. Этот алгоритм основывается на ручном заполнении таблиц маршрутизации.
- Простая маршрутизация. Бывает трех видов: случайная — с посылкой пакетов в случайных направлениях, кроме исходного, лавинная маршрутизация — пакеты передаются во всех направлениях, кроме исходного, маршрутизация по предыдущему опыту — основывается на информации, содержащейся в проходящих пакетах.
- Адаптивная маршрутизация. Наиболее часто используемый вид маршрутизации. Основывается на том, что маршрутизаторы периодически обмениваются информацией о топологии сети (глобальной), которая, кстати, постоянно меняется, при этом учитывается не только сама топология, но и пропускная способность отдельных участков.

Управление сокетами

Socket в переводе означает "гнездо" или "электрическая розетка". Стандартная спецификация Windows Sockets определяет интерфейс в сети TCP/IP, позволяющий приложениям взаимодействовать друг с другом. Можно сказать, что два приложения в сети взаимодействуют в сети посредством "гнезда", к которому они подключены. По своим свойствам сокет напоминает дескриптор файла, только функции управления специфические. Они хранятся в отдельной динамической библиотеке. А для того чтобы использовать эти функции в нашей программе, следует при трансляции подключать библиотеку ws2_32.lib. При описании сокетов все структуры будут описываться так, как это сделано в файле windows.inc из пакета MASM32.

После столь краткого введения приступим к описанию функций управления сокетами, точнее, функций взаимодействия приложений посредством сокетов.

Перед тем как начать работу с библиотекой поддержки сокетов, она должна быть проинициализирована. Для этого используется функция `WSAStartup`.

В случае ошибки функция возвращает ненулевой код. Рассмотрим параметры данной функции.

- 1-й параметр — это двойное слово. Старшее слово не используется. В младшем слове, в первом байте, содержится младшая часть версии, в старшем — старшая часть версии библиотеки.
- 2-й параметр является адресом специальной структуры, которая получает информацию о поддержке сокетов. Поскольку содержимое этой структуры (`WSADATA`) нас интересовать не будет, то достаточно просто зарезервировать для нее нужное количество байтов. Ниже представлена данная структура:

```
WSADATA STRUCT
      wVersion          WORD      ?
      wHighVersion      WORD      ?
      szDescription     BYTE 257 dup (?)
      szSystemStatus    BYTE 129 dup (?)
      iMaxSockets       WORD      ?
      iMaxUdpDg         WORD      ?
      lpVendorInfo      DWORD     ?
WSADATA ENDS
```

Следующая функция `socket` создает сокет. В случае удачного завершения она возвращает дескриптор сокета. В случае ошибки возвращается –1. Функция имеет три параметра.

- 1-й параметр задает семейство протоколов. Для протоколов семейства TCP/IP используется константа `AF_INET = 2`.
- 2-й параметр определяет режим взаимодействия (другими словами, тип создаваемого сокета). Обычно используются две константы[3]: `SOCK_STREAM = 1` (потоковая передача данных) — для установленного соединения, и `SOCK_DGRAM = 2` (передача на основе дейтограмм) — без установленного соединения.
- 3-й параметр задает протокол транспортного уровня. Если задать данный параметр равным нулю, то система будет сама выбирать протокол, исходя из первых двух параметров функции.

Для того чтобы обратиться с запросом к программе-серверу, используется функция `connect`. Функция возвращает 0 при успешном завершении. Она имеет три параметра:

- 1-й параметр должен содержать ранее созданный сокет (дескриптор) — значение, возвращаемое функцией `socket`;

[3] Существуют и другие константы, например, `SOCK_SEQPACKET`, `SOCK_RAW` и др.

- 2-й параметр должен представлять адрес некоторой структуры, содержащей адрес программы-сервера. Данная структура будет подробно рассмотрена далее;
- 3-й параметр — длина структуры.

Обратимся к упомянутой выше структуре. Вот она:

```
sockaddr_in STRUCT
      sin_family  WORD      ?
      sin_port    WORD      ?
      sin_addr    in_addr <>
      sin_zero    BYTE 8 dup (?)
sockaddr_in ENDS
```

Поле `sin_family` структуры должно содержать семейство протоколов, т. е. например `AF_INET = 2`. Поле `sin_port` должно содержать номер *порта* приложения.

> **ЗАМЕЧАНИЕ**
>
> Следует быть внимательным при выборе порта, поскольку многие порты зарезервированы различными службами: HTTP, FTP и т. д. Эти порты распределяются и контролируются центром IANA — Internet Assigned Numbers Authority. Все номера портов можно разделить на три категории:
>
> - 0—1023 — зарезервировано для стандартных служб;
> - 1024—49151 — могут использоваться пользовательскими программами;
> - 49152—65553 — называются динамическими или частными.

Как видим, структура `sockaddr_in` содержит внутри себя еще структуру, используемую для хранения IP-адреса в четырехбайтовом виде:

```
in_addr STRUCT
      S_un ADDRESS_UNION <>
in_addr ENDS
```

которая, фактически, представляет собой объединение:

```
ADDRESS_UNION UNION
      S_un_b S_UN_B <>
      S_un_w S_UN_W <>
      S_addr DWORD ?
ADDRESS_UNION ENDS
```

И, наконец:

```
S_UN_B STRUCT
      s_b1 BYTE ?
      s_b2 BYTE ?
      s_b3 BYTE ?
      s_b4 BYTE ?
```

```
S_UN_B ENDS
S_UN_W STRUCT
      s_w1 WORD ?
      s_w2 WORD ?
S_UN_W ENDS
```

Столь длинные определения в действительности не представляют абсолютно никакой сложности, отражая всего лишь факт, что адрес `sin_addr` может быть задан тремя разными способами. Далее в примерах станет ясно, как пользоваться этими структурами.

Функция `listen`. Эта функция переводит сокет в состояние, в котором он слушает внешние вызовы. Функция возвращает 0, если все нормально. Параметры функции:

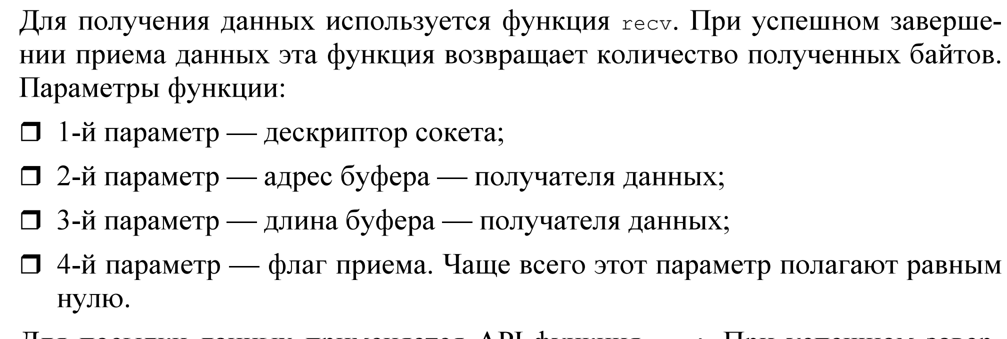
- 1-й параметр — дескриптор сокета;
- 2-й параметр определяет максимальную длину очереди входящих запросов. Стандартное значение равно 5.

Функция `accept`. Используется для приема запросов программ-клиентов на установление связи, которые они осуществляют с помощью функции `connect`. Функция `accept` должна предшествовать функции `listen`, которая организует очередь. Функция извлекает самый первый запрос на соединение и возвращает дескриптор сокета, который будет использоваться для обмена данными с вышедшим на связь клиентом. Если очередь запроса пуста, то функция переходит в состояние ожидания. Рассмотрим параметры функции:

- 1-й параметр — дескриптор сокета, через который данная программа получает запрос;
- 2-й параметр является адресом структуры `sockaddr_in`, которая получит информацию о соединении;
- 3-й параметр — размер структуры, определяемый вторым параметром.

Функция `bind`. Данная функция осуществляет подключение сокета к коммуникационной среде. При успешном завершении функция возвращает 0, в противном случае –1. Параметры функции:

- 1-й параметр — дескриптор связываемого сокета;
- 2-й параметр — указатель на структуру `sockaddr_in`. Предварительно она должна быть заполнена. Поле `sin_family` должно быть равно `AF_INET = 2`. Поле `sin_addr.s_addr` должно быть равно `INADDR_ANY = 0`. В поле `port` должен быть указан номер порта, например 2000;
- 3-й параметр — длина структуры, на которую указывает второй параметр.



Для получения данных используется функция `recv`. При успешном завершении приема данных эта функция возвращает количество полученных байтов. Параметры функции:

- 1-й параметр — дескриптор сокета;
- 2-й параметр — адрес буфера — получателя данных;
- 3-й параметр — длина буфера — получателя данных;
- 4-й параметр — флаг приема. Чаще всего этот параметр полагают равным нулю.

Для посылки данных применяется API-функция `send`. При успешном завершении функция возвращает количество переданных байтов. Параметры функции:

- 1-й параметр — дескриптор используемого сокета;
- 2-й параметр — адрес буфера, где содержатся передаваемые данные;
- 3-й параметр — длина буфера;
- 4-й параметр — флаг, обычно равен 0.

Для закрытия ранее созданного сокета предназначена функция `closesocket`. Единственным параметром этой функции является дескриптор сокета.

Для экстренного закрытия сокета используется функция `shutdown`. Первый параметр этой функции равен дескриптору закрываемого сокета. Второй параметр может принимать следующие значения: 0 — сбросить и далее не принимать данные для чтения из сокета, 1 — сбросить и далее не отправлять передаваемые данные, 2 — сбросить все данные.

Кроме перечисленных выше функций при работе с сокетами полезными окажутся следующие функции.

Функция `gethostname`. Эта функция используется для получения имени локального компьютера. Первый параметр функции — это буфер, куда будет помещено имя. Второй параметр — длина буфера.

Функция `gethostbyname`. Используется для получения информации об удаленном в сети компьютере. Единственным ее параметром является указатель на имя в сети. Сама же функция возвращает указатель на структуру *(см. далее)* или 0, если произошла ошибка. Рассмотрим возвращаемую структуру.

```
hostent STRUCT
      h_name   DWORD  ?
      h_alias  DWORD  ?
      h_addr   WORD   ?
      h_len    WORD   ?
      h_list   DWORD  ?
hostent ENDS
```

Поля структуры:

- h_name — адрес, куда помещается официальное имя узла;
- h_alias — указатель на массив дополнительных имен. Имена отделяются друг от друга 0, в конце массива два нуля;
- h_addr — тип адреса, имеет значение 2 (AF_INET);
- h_len — длина адреса узла;
- h_list — указывает на массив, содержащий IP-адреса данного узла, отделенные друг от друга кодом 0. В конце массива два нуля. IP-адрес представляет собой просто последовательность четырех байтов.

Весьма полезной может быть функция inet_addr. Она переводит строку — IP-адрес в 32-битное число. Единственным параметром функции является адрес строки IP-адреса. Возвращаемое 32-битное число содержит четыре байта, т. е. компоненты IP-адреса. У данной функции есть и обратная функция — inet_ntoa, которая переводит четырехбайтовый адрес в строковый эквивалент.

Пример простейшего клиента и сервера

В данном разделе приведен пример, состоящий из двух программ: клиента и сервера. Они взаимодействуют друг с другом по протоколу TCP/IP. Тексты программ представлены в листингах 3.4.5 (сервер) и 3.4.6 (клиент). Здесь представлен самый простой вариант такой системы, которая, однако, содержит все основные механизмы взаимодействия приложений с помощью сокетов. Сервер ждет вызова клиента и при обращении к нему клиента посылает ему строку, которую тот выводит на консольный экран. В ответ клиент также посылает серверу сообщение, которое сервер также выводит на консоль. Ожидание сервера происходит в цикле, т. е. он может откликнуться на 10 вызовов клиента, идущих один за другим. Для соединения с сервером клиент должен знать сетевое имя компьютера (в листинге 3.4.6 имя компьютера хранится в переменной comp), где запущена программа-сервер. Предварительно он определяет IP-адрес по имени и выводит его на консоль.

Листинг 3.4.5. Программа-сервер, принимающая вызов от клиента

```
;программа server.asm
.586P
;плоская модель
.MODEL FLAT, stdcall
;константы
```

```
STD_OUTPUT_HANDLE equ -11
;прототипы внешних процедур
EXTERN  shutdown@8:NEAR
EXTERN  recv@16:NEAR
EXTERN  send@16:NEAR
EXTERN  accept@12:NEAR
EXTERN  listen@8:NEAR
EXTERN  bind@12:NEAR
EXTERN  closesocket@4:NEAR
EXTERN  socket@12:NEAR
EXTERN  CharToOemA@8:NEAR
EXTERN  WSAStartup@8:NEAR
EXTERN  wsprintfA:NEAR
EXTERN  GetLastError@0:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  GetStdHandle@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\ws2_32.lib
;-----------------------------------------------
WSADATA STRUCT
wVersion        WORD   ?
wHighVersion    WORD   ?
szDescription   BYTE 257 dup (?)
szSystemStatus  BYTE 129 dup (?)
iMaxSockets     WORD   ?
iMaxUdpDg       WORD   ?
lpVendorInfo    DWORD  ?
WSADATA ENDS
;-----------------------------------------------
S_UN_B STRUCT
      s_b1 BYTE 0
      s_b2 BYTE 0
      s_b3 BYTE 0
      s_b4 BYTE 0
S_UN_B ENDS
S_UN_W STRUCT
      s_w1 WORD 0
      s_w2 WORD 0
S_UN_W ENDS
ADDRESS_UNION UNION
      s_u_b S_UN_b <>
      s_u_w S_UN_w <>
```

```
      s_addr DWORD 0
ADDRESS_UNION ENDS
in_addr STRUCT
      s_un ADDRESS_UNION <>
in_addr ENDS
sockaddr_in STRUCT
      sin_family    WORD      0
      sin_port      WORD      0
      sin_addr      in_addr <>
      sin_zero      BYTE 8 dup (0)
sockaddr_in ENDS
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      HANDL DD ?
      LENS  DD ?
      ERRS  DB "Error %u ",0
;-----------------------------------------------
      S1    DD ?
      S2    DD ?
      LEN   DD ?
      BUF   DB 100 DUP(0)
      BUF1  DB 100 DUP(0)
      txt   DB 'Вызов принят. Вышлите подтверждение.',0
      msg   DB 'Сервер завершил работу',0
      sin1  sockaddr_in <0>
      sin2  sockaddr_in <0>
      wsd   WSADATA <0>
      len1  DD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;определить дескриптор консоли вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;активизировать библиотеку сокетов
      PUSH OFFSET wsd
      MOV  EAX,0
      MOV  AX,0202H
      PUSH EAX
      CALL WSAStartup@8
      CMP  EAX,0
      JZ   NO_ER1
      CALL ERRO
```

```
      JMP  EXI
NO_ER1:
;создать сокет
      PUSH 0
      PUSH 1  ;SOCK_STREAM
      PUSH 2  ;AF_INET
      CALL socket@12
      CMP  EAX, NOT 0
      JNZ  NO_ER2
      CALL ERRO
      JMP  EXI
NO_ER2:
      MOV  s1,EAX
;подключить сокет
      MOV  sin1.sin_family,2           ;AF_INET
      MOV  sin1.sin_addr.s_un.s_addr,0 ;INADDR_ANY
      MOV  sin1.sin_port,2000          ;номер порта
      PUSH sizeof(sockaddr_in)
      PUSH OFFSET sin1
      PUSH s1
      CALL bind@12
      CMP  EAX,0
      JZ   NO_ER3
      CALL ERRO
      JMP  CLOS
NO_ER3:
;перевести сокет в состояние "слушать"
      MOV  ESI,10
      PUSH 5
      PUSH s1
      CALL listen@8
      CMP  EAX,0
      JZ   NO_ER4
      CALL ERRO
      JMP  CLOS
NO_ER4:
      MOV  len1,sizeof(sockaddr_in)
;ждем запроса от клиента
      PUSH OFFSET len1
      PUSH OFFSET sin2
      PUSH s1
      CALL accept@12
      MOV  s2,EAX
;запрос пришел - посылаем информацию
      PUSH 0
      PUSH OFFSET txt
```

```
      CALL lstrlenA@4
      PUSH EAX
      PUSH OFFSET txt
      PUSH s2
      CALL send@16
;ждем ответа
      PUSH 0
      PUSH 100
      PUSH OFFSET buf
      PUSH s2
      CALL recv@16 ;в EAX - длина сообщения
;в начале перекодировка
      PUSH OFFSET buf1
      PUSH OFFSET buf
      CALL CharToOemA@8
;теперь вывод
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
;закрыть связь
      PUSH 0
      PUSH s2
      CALL shutdown@8
;закрыть сокет
      PUSH S2
      CALL closesocket@4
      DEC  ESI
      JNZ  NO_ER4
;конец цикла
      PUSH OFFSET buf1
      PUSH OFFSET msg
      CALL CharToOemA@8
;теперь вывод
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
CLOS:
      PUSH S1
      CALL closesocket@4
EXI:
;выход происходит по завершению всех служб
      PUSH 0
      CALL ExitProcess@4
;вывести строку (в конце перевод строки)
;EAX - на начало строки
;EDI - с переводом строки или без
```

```
WRITE  PROC
      PUSH  ESI
;получить длину параметра
      PUSH  EAX
      PUSH  EAX
      CALL  lstrlenA@4
      MOV   ESI,EAX
      POP   EBX
      CMP   EDI,1
      JNE   NO_ENT
;в конце - перевод строки
      MOV  BYTE PTR [EBX+ESI],13
      MOV  BYTE PTR [EBX+ESI+1],10
      MOV  BYTE PTR [EBX+ESI+2],0
      ADD  EAX,2
NO_ENT:
;вывод строки
      PUSH 0
      PUSH OFFSET LENS
      PUSH EAX
      PUSH EBX
      PUSH HANDL
      CALL WriteConsoleA@20
      POP  ESI
      RET
WRITE  ENDP
;процедура вывода номера ошибки
ERRO  PROC
      CALL GetLastError@0
      PUSH EAX
      PUSH OFFSET ERRS
      PUSH OFFSET BUF1
      CALL wsprintfA
      ADD  ESP,12
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
      RET
ERRO  ENDP
_TEXT ENDS
END START
```

Трансляция программы server.asm (см. листинг 3.4.5):

```
ml /c /coff  server.asm
link /subsystem:console server.obj
```

Комментарий к программе из листинга 3.4.5.

- Программа server.asm ожидает выход на нее клиентских программ. Она слушает порт с номером 2000. Ожидание осуществляется функцией `accept`. Функция возвращает управление при попытке соединиться с данным сервером. Если соединение произошло, то функция возвращает вновь созданный сокет, на котором и будет осуществляться взаимодействие с программой-клиентом.
- В работе функции `accept` есть одна тонкость. Она возвращает управление сразу после получения сообщения от клиента. Это еще не гарантирует, что соединение установлено, особенно это справедливо для глобальной сети. Следовательно, строя взаимодействие по данной схеме, следует предусмотреть дополнительные действия для подтверждения соединения.
- В программе реализована простейшая схема взаимодействия, когда сервер может взаимодействовать в данный момент лишь с одним клиентом. Для возможной работы с несколькими клиентами необходим другой подход. В *главе 2.8* мы уже говорили о подобном механизме: при получении сообщения от клиента создается новый поток, которому передается вновь созданный сокет. Это поток в дальнейшем будет полностью заниматься с данным клиентом. Основной же поток снова переходит к ожиданию новых клиентов.

Листинг 3.4.6. Программа-клиент, вызывающая программу-сервер

```
;программа klient.asm
.586P
;плоская модель
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
;прототипы внешних процедур
EXTERN  connect@12:NEAR
EXTERN  gethostbyname@4:NEAR
EXTERN  shutdown@8:NEAR
EXTERN  recv@16:NEAR
EXTERN  send@16:NEAR
EXTERN  accept@12:NEAR
EXTERN  listen@8:NEAR
EXTERN  bind@12:NEAR
EXTERN  closesocket@4:NEAR
EXTERN  socket@12:NEAR
EXTERN  CharToOemA@8:NEAR
EXTERN  WSAStartup@8:NEAR
```

```
EXTERN  wsprintfA:NEAR
EXTERN  GetLastError@0:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  GetStdHandle@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\ws2_32.lib
;-----------------------------------------------
WSADATA STRUCT
wVersion        WORD   ?
wHighVersion    WORD   ?
szDescription   BYTE 257 dup (?)
szSystemStatus  BYTE 129 dup (?)
iMaxSockets     WORD   ?
iMaxUdpDg       WORD   ?
lpVendorInfo    DWORD  ?
WSADATA ENDS
;-----------------------------------------------
S_UN_B STRUCT
      s_b1 BYTE 0
      s_b2 BYTE 0
      s_b3 BYTE 0
      s_b4 BYTE 0
S_UN_B ENDS
S_UN_W STRUCT
      s_w1 WORD 0
      s_w2 WORD 0
S_UN_W ENDS
ADDRESS_UNION UNION
      s_u_b S_UN_b <>
      s_u_w S_UN_w <>
      s_addr DWORD 0
ADDRESS_UNION ENDS
in_addr STRUCT
      s_un ADDRESS_UNION <>
in_addr ENDS
sockaddr_in STRUCT
      sin_family  WORD      0
      sin_port    WORD      0
      sin_addr    in_addr <>
      sin_zero    BYTE 8 dup (0)
sockaddr_in ENDS
```

```
hostent STRUCT
      h_name    DWORD ?
      h_alias   DWORD ?
      h_addr    WORD  ?
      h_len     WORD  ?
      h_list    DWORD ?
hostent ENDS
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      HANDL DD ?
      LENS  DD ?
      ERRS  DB "Error %u ",0
      IP    DB "IP address %hu.%hu.%hu.%hu",0
      IPA   DD ?
      S1    DD ?
      comp  DB "dom2",0  ;имя компьютера, где располагается сервер
      txt   DB "Подтверждаю вызов",0
      txt1  DB "Адрес компьютера",0
      LEN   DD ?
      sin2  sockaddr_in <0>
      hp    hostent <0>
      BUF   DB  100 DUP(0)
      BUF1  DB  100 DUP(0)
      wsd   WSADATA <0>
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;определить дескриптор консоли вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;активизировать библиотеку сокетов
      PUSH OFFSET wsd
      MOV  EAX,0
      MOV  AX,0202H
      PUSH EAX
      CALL WSAStartup@8
      CMP  EAX,0
      JZ   NO_ER1
      CALL ERRO
      JMP  EXI
NO_ER1:
;определить адрес сервера (хоста) по его имени
      PUSH OFFSET comp
```

```
      CALL gethostbyname@4
      CMP  EAX,0
      JNZ  NO_ER2
      CALL ERRO
      JMP  EXI
NO_ER2:
;вывести адрес
      MOV  EBX,[EAX+12]  ;h_list в структуре hostent
      MOV  EDX,[EBX]
      MOV  EDX,[EDX]
      MOV  IPA,EDX
      SHR  EDX,24
      AND  EDX,000000FFH
      PUSH EDX
      MOV  EDX,IPA
      SHR  EDX,16
      AND  EDX,000000FFH
      PUSH EDX
      MOV  EDX,IPA
      SHR  EDX,8
      AND  EDX,000000FFH
      PUSH EDX
      MOV  EDX,IPA
      AND  EDX,000000FFH
      PUSH EDX
      PUSH OFFSET IP
      PUSH OFFSET BUF1
      CALL wsprintfA
      ADD  ESP,24
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
      MOV  EDX,IPA
      MOV  sin2.sin_addr.s_un.s_addr,EDX
      MOV  sin2.sin_port,2000
      MOV  sin2.sin_family,2 ;AF_INET
;создать сокет
      PUSH 0
      PUSH 1  ;SOCK_STREAM
      PUSH 2  ;AF_INET
      CALL socket@12
      CMP  EAX, NOT 0
      JNZ  NO_ER3
      CALL ERRO
      JMP  EXI
NO_ER3:
      MOV  s1,EAX
```

```
;попытка соединиться с сервером
      PUSH sizeof(sockaddr_in)
      PUSH OFFSET sin2
      PUSH s1
      CALL connect@12
      CMP  EAX, 0
      JZ   NO_ER4
      CALL ERRO
      JMP  CLOS
NO_ER4:
;ждем информацию
      PUSH 0
      PUSH 100
      PUSH OFFSET buf
      PUSH s1
      CALL recv@16 ;в EAX - длина сообщения
;в начале перекодировка
      PUSH OFFSET buf1
      PUSH OFFSET buf
      CALL CharToOemA@8
;теперь вывод
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
;посылаем информацию
      PUSH 0
      PUSH OFFSET txt
      CALL lstrlenA@4
      PUSH EAX
      PUSH OFFSET txt
      PUSH s1
      CALL send@16
CLOS:
      PUSH S1
      CALL closesocket@4
EXI:
;выход происходит по завершению всех служб
      PUSH 0
      CALL ExitProcess@4
;вывести строку (в конце перевод строки)
;EAX - на начало строки
;EDI - с переводом строки или без
WRITE   PROC
;получить длину параметра
      PUSH  EAX
```

```
      PUSH  EAX
      CALL  lstrlenA@4
      MOV   ESI,EAX
      POP   EBX
      CMP   EDI,1
      JNE   NO_ENT
;в конце - перевод строки
      MOV  BYTE PTR [EBX+ESI],13
      MOV  BYTE PTR [EBX+ESI+1],10
      MOV  BYTE PTR [EBX+ESI+2],0
      ADD  EAX,2
NO_ENT:
;вывод строки
      PUSH 0
      PUSH OFFSET LENS
      PUSH EAX
      PUSH EBX
      PUSH HANDL
      CALL WriteConsoleA@20
      RET
WRITE  ENDP
;процедура вывода номера ошибки
ERRO    PROC
CALL GetLastError@0
PUSH EAX
      PUSH OFFSET ERRS
      PUSH OFFSET BUF1
      CALL wsprintfA
      ADD  ESP,12
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
      RET
ERRO  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.4.6:

```
ml /c /coff /DMASM klient.asm
link /subsystem:console klient.obj
```

Комментарий к программе из листинга 3.4.6.

- Прежде чем обратиться к серверу, клиент должен знать IP-адрес узла, где расположена программа-сервер. Для этого используется функция `gethostbyname`.

Имя компьютера, где располагается программа-сервер, хранится в переменной `comp`. Данная функция при успешном поиске возвращает указатель на структуру `HOSTENT`, о которой мы уже говорили ранее и определение которой мы помещаем в нашу программу, хотя переменную такого типа мы здесь непосредственно не используем. Чтобы облегчить вам понимание того, как мы потом "достаем" IP-адрес, приведу соответствующие строки из программы.

```
MOV  EBX,[EAX+12] ;h_list в структуре hostent
MOV  EDX,[EBX]
MOV  EDX,[EDX]
MOV  IPA,EDX
```

Обратите внимание, что смещение на 12 байтов от начала структуры — это как раз поле `h_list`. Поле же это есть указатель на строку байтов. Это строка состоит из четверок байтов, каждая из которых представляет собой IP-адрес узла — ведь адресов у узла может быть несколько. Признаком конца этой последовательности является нулевой код. Нас же интересует только первая четверка, которую мы считываем командой `MOV EDX,[EDX]`. При этом младший байт — это крайний левый байт в IP-адресе.

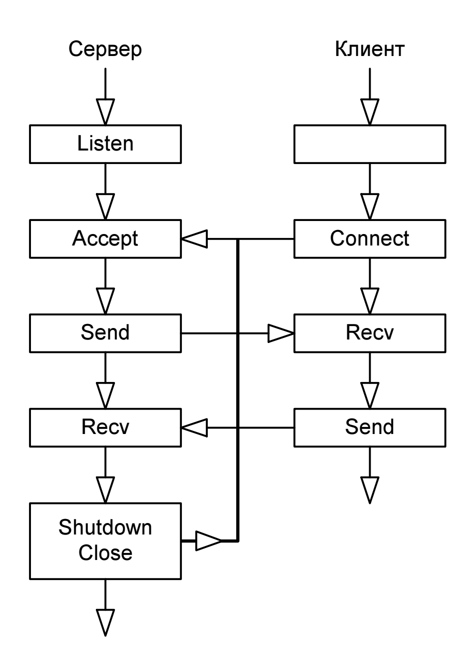
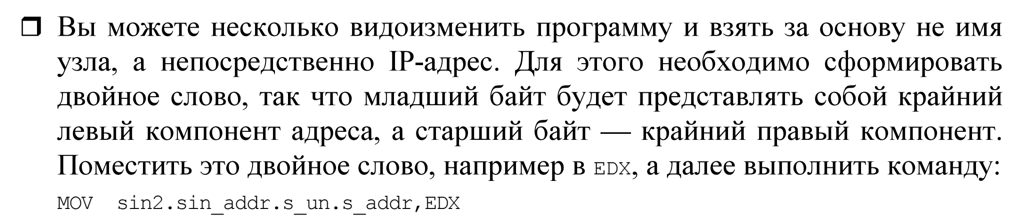


Рис. 3.4.4. Схема взаимодействия клиента и сервера, тексты программ которых представлены в листингах 3.4.5 и 3.4.6



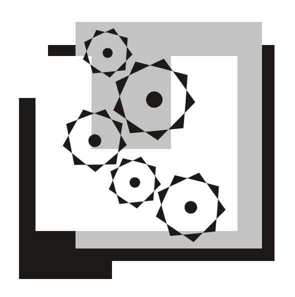
- Вы можете несколько видоизменить программу и взять за основу не имя узла, а непосредственно IP-адрес. Для этого необходимо сформировать двойное слово, так что младший байт будет представлять собой крайний левый компонент адреса, а старший байт — крайний правый компонент. Поместить это двойное слово, например в `EDX`, а далее выполнить команду:

```
MOV  sin2.sin_addr.s_un.s_addr,EDX
```

как это было сделано в программе.

Завершая рассмотрение примеров работы с сокетами, хочу обратить внимание читателей на рис. 3.4.4, где представлена схема взаимодействия программ server.asm и klient.asm.

В заключение главы хочу предложить читателям следующее простое упражнение: видоизмените программу из листинга 3.4.3 так, чтобы для каждого из найденных сетевых компьютеров она определяла IP-адрес и выводила этот адрес вместе с именем компьютера.



Глава 3.5

Разрешение некоторых проблем программирования в Windows

Признаться, данная глава стала для меня некоторым компромиссом. Вопросов, возникающих при программировании в Windows, так много, что не хотелось бы их опускать. А если писать на каждый вопрос целую главу — книга станет непомерно большой. И я решил отвести одну главу, где постараюсь, насколько возможно кратко, осветить ряд интересных вопросов. Глава будет построена в виде вопросов гипотетического собеседника и ответов на них вашего покорного слуги. Итак, приступим.

Вопрос: как сделать, чтобы при минимизации окна его значок помещался на системную панель?

Эта проблема решается с использованием системной функции `Shell_NotifyIcon`. Причем решение это столь просто и прозрачно, что приходится удивляться тому, что большинство для этих целей привлекают различные библиотеки. Вот параметры этой функции.

- 1-й параметр определяет действие, которое необходимо произвести:
  - `NIM_ADD equ 0h` — добавить пиктограмму на системную панель;
  - `NIM_MODIFY equ 1h` — удалить пиктограмму;
  - `NIM_DELETE equ 2h` — модифицировать пиктограмму;
  - `NIM_SETFOCUS equ 3h` — возвратить фокус системной панели. Необходимость в данном сообщении может возникнуть, например, в том случае, если пользователь отменил использование меню, нажав клавишу <Esc>;
  - `NIM_SETVERSION equ 4h` — инструкция для системной панели вести себя согласно номеру версии, которая хранится в поле `uVersion` (см. второй параметр).

- 2-й параметр — указатель на структуру, где содержится информация, необходимая для реализации указанного действия.

```
NOTI_ICON STRUC
      cbSize             DD      ?
      hWnd               DD      ?
      uID                DD      ?
      uFlags             DD      ?
      uCallbackMessage   DD      ?
      hIcon              DD      ?
      szTip              DB 128 DUP     (?)
      dwState            DD      ?
      dwStateMask        DD      ?
      szInfo             DB  255 DUP    (?)
      UN                 UN1 <>
      szInfoTitle        DB      64 DUP (?)
      dwInfoFlags        DD      ?
      guidItem           DB      16 DUP (?)
NOTI_ICON ENDS
```

Структура `UN1`:

```
UN1  UNION
      uTimeout      DD  ?
      uVersion      DD  ?
UN1  ENDS
```

Поля:

- `cbSize` — размер структуры;
- `hWnd` — дескриптор окна, куда будет посылаться сообщение *(см. далее)*;
- `uID` — идентификатор пиктограммы. Для одного окна можно определить несколько пиктограмм;
- `uFlags` — комбинация следующих флагов:
  - ◊ `NIF_MESSAGE equ 1h` — использовать поле `hIcon`;
  - ◊ `NIF_ICON equ 2h` — использовать поле `uCallbackMessage`;
  - ◊ `NIF_TIP equ 4h` — использовать поле `szTip`;
  - ◊ `NIF_STATE equ 8h` — использовать поля `dwState` и `dwStateMask`;
  - ◊ `NIF_INFO equ 10h` — используются поля `szInfo`, `uTimeout`, `szInfoTitle`, `dwInfoFlags`;
  - ◊ `NIF_GUID equ 20h` — зарезервировано;
- `uCallbackMessage` — сообщение, которое приходит в окно, определяемое дескриптором `hWnd`, в случае возникновения некоего события вблизи пиктограммы на системной панели. Значение сообщения должно быть

больше, чем 1024. При приходе этого сообщения WPARAM содержит идентификатор пиктограммы, а LPARAM — событие, т. е. то, что произошло с пиктограммой. Например, если курсор мыши наведен на пиктограмму, то LPARAM будет содержать сообщение WM_MOUSEMOVE;

- hIcon — дескриптор пиктограммы, над которой производится действие: добавление, удаление или модификация;
- SzTip — текст стандартной подсказки. В конце обязательный 0;
- dwState — принимает два значения: NIS_HIDDEN — пиктограмма скрыта, NIS_SHAREDICON — пиктограмма является разделяемым ресурсом;
- dwSateMask — указывает, какие биты поля dwState задействованы;
- szInfo — текст всплывающей подсказки (новый вид для значков на системной панели);
- uTimeout — время жизни для всплывающей подсказки;
- uVersion — показывает версию функции. Значение 0 означает, что работает старая версия (до Windows 2000), 3 — новая версия;
- szInfoTitle — здесь содержится заголовок для всплывающей подсказки (новая версия);
- dwInfoFlags — флаг для помещения во всплывающую подсказку:
  - ◊ NIIF_ERROR equ 3h — пиктограмма ошибки;
  - ◊ NIIF_INFO equ 1h — информационная пиктограмма;
  - ◊ NIIF_NONE equ 0h — пиктограмма отсутствует;
  - ◊ NIIF_USER equ 4h — использовать идентификатор hIcon для отображения во всплывающей подсказке;
  - ◊ NIIF_WARNING equ 2h — предупреждающая пиктограмма;
  - ◊ NIIF_ICON_MASK equ 0fh — зарезервировано;
  - ◊ NIIF_NOSOUND equ 10h — зарезервировано.

Замечание

В ранних версиях функции Shell_NotifyIcon структура NOTI_ICON имела несколько иной формат. Последним полем в ней было поле szTip, которое имело размер 64 байта. Но вы и сейчас можете использовать такую половинную структуру, если вам нет надобности во всплывающих подсказках нового типа. Не забывайте только правильно указать длину такой структуры в поле cbSize.

Давайте рассмотрим, как работает весь механизм. В случае минимизации окна на его функцию приходит сообщение WM_SIZE. Причем wParam должен содержать значение SIZE_MINIMIZED. Вот тогда-то и следует воспользоваться функ-

цией `Shell_NotifyIcon`, которая поместит пиктограмму на системную панель. Любые же события с мышью, когда ее курсор находится на пиктограмме, вызовут приход в функцию окна сообщения `uCallbackMessage`, которое, естественно, мы сами и определили. При этом младшее слово `wParam` будет содержать идентификатор пиктограммы, а по младшему слову `lParam` можно определить тип события. В листинге 3.5.1 вы увидите, как все это работает (см. рис. 3.5.1, где представлен новый вид всплывающей подсказки).

Листинг 3.5.1. Демонстрация процедуры помещения пиктограммы на системную панель

```
//файл tray.rc
//определение констант
#define WS_SYSMENU      0x00080000L
#define WS_MINIMIZEBOX  0x00020000L
#define WS_MAXIMIZEBOX  0x00010000L
#define DS_3DLOOK       0x0004L
//идентификаторы
#define IDI_ICON1 1
//определили пиктограмму
IDI_ICON1 ICON "ico1.ico"
//определение диалогового окна
DIAL1 DIALOG 0, 0, 250, 110
STYLE WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
DS_3DLOOK
CAPTION "Поместить пиктограмму на системную панель"
FONT 8, "Arial"
{
}
;файл tray.inc
;константы
NIM_ADD             equ 0h  ; добавить пиктограмму на системную панель
NIM_MODIFY          equ 1h  ; удалить пиктограмму
NIM_DELETE          equ 2h  ; модифицировать пиктограмму
NIF_MESSAGE         equ 1h  ; использовать поле hIcon
NIF_ICON            equ 2h  ; использовать поле uCallbackMessage
NIF_TIP             equ 4h  ; использовать поле szTip
NIF_INFO            equ 10h
NIF_STATE           equ 8h
FLAG                equ  NIF_INFO or NIF_ICON or NIF_MESSAGE or NIF_TIP
NIM_SETVERSION      equ 4h
NOTIFYICON_VERSION  equ 3h
NIIF_INFO           equ 1h
NIIF_ERROR          equ 3h
SIZE_MINIMIZED      equ 1h
```

```
SW_HIDE   equ 0
SW_SHOWNORMAL       equ 1
;сообщение приходит при закрытии окна
WM_CLOSE            equ 10h
WM_INITDIALOG       equ 110h
WM_SIZE             equ 5h
WM_LBUTTONDBLCLK    equ 203h
WM_LBUTTONDOWN      equ 201h
;прототипы внешних процедур
EXTERN  ShowWindow@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  lstrcpyA@8:NEAR
EXTERN  Shell_NotifyIconA@8:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  SendDlgItemMessageA@20:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;структура для функции Shell_NotifyIcon
UN1  UNION
      uTimeout  DD  0
      uVersion  DD  0
UN1  ENDS

NOTI_ICON STRUC
      cbSize            DD   0
      hWnd              DD   0
      uID               DD   0
      uFlags            DD   0
      uCallbackMessage  DD   0
      hIcon             DD   ?
      szTip             DB 128 DUP(?)
      dwState           DD   ?
      dwStateMask       DD   ?
      szInfo            DB 256 DUP (?)
      UN   UN1          <>
```

```
      szInfoTitle       DB  64 DUP (?)
      dwInfoFlags       DD   ?
      guidItem          DB  16 DUP (?)
NOTI_ICON ENDS
;файл tray.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include tray.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\shell32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG    MSGSTRUCT <?>
      HINST  DD 0 ;дескриптор приложения
      PA     DB "DIAL1",0
      NOTI   NOTI_ICON <0>
      TIP    DB "Новый вид подсказки. Пример использования функции "
             DB "Shell_NotifyIcon.",0
      TIP1   DB "Заголовок новой подсказки",0
      TIP2   DB "Старый вид подсказки.",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
;--------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  [HINST]
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
;сообщение об ошибке
KOL:
;---------------------------------
      PUSH  0
```

```
      CALL  ExitProcess@4
;--------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;----------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP  FINISH
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  L2
      JMP  FINISH
L2:
      CMP  DWORD PTR [EBP+0CH],WM_SIZE
      JNE  L3
      CMP  DWORD PTR [EBP+10H],SIZE_MINIMIZED
      JNE  L3
;здесь работа по установке пиктограммы на системную панель
      MOV  NOTI.cbSize,size noti
      MOV  EAX,DWORD PTR [EBP+8H]
      MOV  NOTI.hWnd,EAX
      MOV  NOTI.uFlags,FLAG
      MOV  NOTI.uID,12  ; идентификатор пиктограммы
      MOV  NOTI.uCallbackMessage,2000   ; сообщение
;загрузить пиктограмму из ресурсов
      PUSH 1
      PUSH HINST
      CALL LoadIconA@8
      MOV  NOTI.hIcon,EAX
;скопировать текст всплывающего сообщения (старый вариант)
      PUSH OFFSET TIP2
      PUSH OFFSET NOTI.szTip
      CALL lstrcpyA@8
```

```
;скопировать текст всплывающего сообщения
      PUSH OFFSET TIP
      PUSH OFFSET NOTI.szInfo
      CALL lstrcpyA@8
;скопировать заголовок текста всплывающего сообщения
      PUSH OFFSET TIP1
      PUSH OFFSET NOTI.szInfoTitle
      CALL lstrcpyA@8
;промежуток времени
      MOV  NOTI.UN.uTimeout,30000
;тип всплывающей подсказки
      MOV  NOTI.dwInfoFlags,NIIF_INFO
;поместить пиктограмму
      PUSH OFFSET NOTI
      PUSH NIM_ADD
      CALL Shell_NotifyIconA@8
;спрятать минимизированное окно
      PUSH SW_HIDE
      PUSH DWORD PTR [EBP+08H]
      CALL ShowWindow@8
;установить версию функции Shell_NotifyIcon
      MOV  NOTI.UN.uVersion,NOTIFYICON_VERSION
      PUSH OFFSET NOTI
      PUSH NIM_SETVERSION
      CALL Shell_NotifyIconA@8
      JMP  FINISH
L3:
;сообщение от пиктограммы на системной панели?
      CMP  DWORD PTR [EBP+0CH],2000
      JNE  FINISH
;идентификатор нашей пиктограммы?
      CMP  WORD PTR [EBP+10H],12
      JNE  FINISH
;что произошло? двойной щелчок?
      CMP  WORD PTR [EBP+14H],WM_LBUTTONDBLCLK
      JNE  FINISH
;заполнить структуру
      MOV  NOTI.cbSize,size noti
      MOV  EAX,DWORD PTR [EBP+8H]
      MOV  NOTI.hWnd,EAX
      MOV  NOTI.uFlags,NIF_ICON
      MOV  NOTI.uID,12  ; идентификатор пиктограммы
      MOV  NOTI.uCallbackMessage,3000   ; сообщение
;удалить пиктограмму
      PUSH OFFSET NOTI
      PUSH NIM_DELETE
```

```
      CALL Shell_NotifyIconA@8
;восстановить окно
      PUSH SW_SHOWNORMAL
      PUSH DWORD PTR [EBP+08H]
      CALL ShowWindow@8
FINISH:
      MOV  EAX,0
      POP  EDI
      POP  ESI
      POP  EBX
      POP  EBP
      RET  16
WNDPROC  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.5.1:

```
ml /c /coff tray.asm
rc tray.rc
link /subsystem:windows tray.obj tray.res
```

Замечу, что новая подсказка к пиктограмме на системной панели появляется сразу, как только мы сворачиваем окно, и исчезает после истечения времени или после щелчка по ней. После этого при наведении на пиктограмму курсора мыши будет появляться обычная подсказка.

В связи с программой из листинга 3.5.1 хочу особо акцентировать ваше внимание на сообщении `WM_SIZE`. Весьма полезное сообщение, я вам скажу. Представьте, что в окне вы расположили какую-то информацию. Если окно допускает изменение размеров, то вам придется решать проблему размещения информации в случае, если размер окна изменился. Так вот, аккуратно все перерисовать и отмасштабировать можно как раз, если использовать данное сообщение. Подчеркну, что сообщение посылается, когда размер окна уже изменился. При этом `WPARAM` содержит признак того, что произошло с окном, а `LPARAM` — новый размер окна (младшее слово — ширина, старшее — высота).

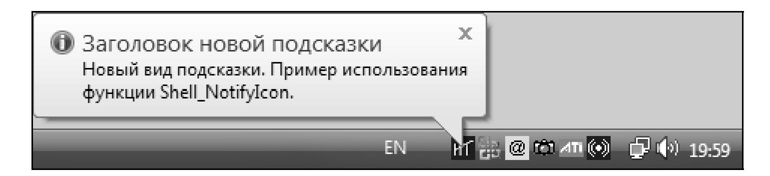


Рис. 3.5.1. Новый вид всплывающей подсказки для значка на системной панели

Вопрос: имеются ли в Windows дополнительные средства, упрощающие файловую обработку?

Да, таким средством, в частности, являются файлы, отображаемые в памяти. Побайтное чтение из файла действительно не всегда удобно. Конечно, можно подойти к этой проблеме несколько иначе. Можно выделить в памяти достаточно большой буфер и прочитать туда файл, а затем работать с ним, как с большим массивом. Так вот, файлы, отображаемые в память, — это нечто похожее, но здесь система берет на себя часть такой работы. Самое замечательное, однако, в этом механизме заключается в том, что файл, спроецированный в память, может быть использован несколькими потоками, а это еще один способ обмена информацией между ними.

У файлов, проецируемых в память, есть довольно существенный недостаток. После отображения их размер не может быть увеличен. В этом случае поступают следующим образом: заранее предполагая новый размер файла, проецируют его на большую область памяти. В действительности файл не загружается в память в прямом смысле, а реализуется в виде отображения на некоторую область виртуальной памяти *(см. главу 3.6)*, страницы которой, разумеется, могут находиться в оперативной памяти, а могут храниться и на диске.

> ***Замечание***
>
> Интересно, но загружаемая в память программа в действительности реализуется в виде отображаемого в память файла. Отдельные части программы, таким образом, оказываются в физической памяти лишь по мере обращения к ним *(см. главу 3.6)*.

Работу с файлами, отображаемыми в память, производят по следующему алгоритму:

1. Открыть (или создать) файл с помощью обычной функции `CreateFile`. Функция, как известно, возвращает дескриптор открытого файла.
2. Создать отображенный файл[1] с помощью функции `CreateFileMapping`. Именно эта функция определяет размер отображения. Эта функция, как и предыдущая, возвращает дескриптор, только не обычного, а отображенного файла.
3. Скопировать файл (или его часть) в созданную область при помощи функции `MapViewOfFile`. Эта функция возвращает указатель (вот оно!) на начало области, где будет расположен файл. После этого руки у вас развя-

---

[1] Возможно, правильнее сказать, что функция создает объект под названием "отображаемый файл".

заны, и вы можете делать с отображенным файлом все, что вам заблагорассудится.

4. При желании можно записать область памяти в файл при помощи функции `FlushViewOfFile`. Разумеется, сбрасываться на диск будет та область памяти, которую мы заказали при помощи функции `CreateFileMapping`. Записывать на диск можно, разумеется, и с помощью обычной функции `WriteFile`.
5. Перед тем как закрывать отображенный файл, следует вначале сделать указатель недействительным. Это осуществляется с помощью функции `UnmapViewOfFile`.
6. Закрыть следует оба дескриптора. Вначале дескриптор, возвращенный функцией `CreateFileMapping`, а затем дескриптор, созданный функцией `CreateFile`.

Как видите, алгоритм работы с отображаемыми файлами весьма прост. Рассмотрим теперь подробнее новые для вас функции.

Функция `CreateFileMapping` возвращает дескриптор отображаемого файла.

- 1-й параметр — дескриптор открытого файла.
- 2-й параметр — атрибут доступа, обычно полагают равным нулю.
- 3-й параметр может принимать одно из следующих значений и должен быть совместим с режимом разделения файла: `PAGE_READONLY`, `PAGE_WRITECOPY`, `PAGE_READWRITE`. Как вы понимаете, этот атрибут определяет защиту отображаемого файла и не должен противоречить атрибуту файла, открытого с помощью функции `CreateFile`.
- 4-й параметр — старшая часть (64-битного значения) размера отображаемого файла.
- 5-й параметр — младшая часть размера отображаемого файла (как вы понимаете, размер может и не совпадать с размером файла). Если оба параметра равны нулю, то размер полагается равным размеру открытого файла (1-й параметр).
- 6-й параметр — имя отображаемого файла. Необходимо только в том случае, если предполагается, что отображаемый файл будет использоваться несколькими процессами. В этом случае повторный вызов функции `CreateFileMapping` другими процессами с тем же именем приведет не к созданию нового отображаемого файла, а к возвращению уже созданного дескриптора.

Функция `MapViewOfFile` возвращает указатель на область памяти, где размещается отображенный файл или его часть.

- 1-й параметр — дескриптор, возвращенный функцией `CreateFileMapping`.

- 2-й параметр определяет операцию, которую мы будем делать. Например, `FILE_MAP_READ` означает только чтение, а `FILE_MAP_WRITE` — чтение и запись.
- 3-й параметр — старшая часть (64-битного значения) смещения в файле, откуда начинается копирование в память.
- 4-й параметр — младшая часть смещения в файле, откуда начинается копирование.
- 5-й параметр определяет количество копируемых байтов. Если вы хотите скопировать весь файл, то задайте три последних параметра равными 0.

Функция `FlushViewOfFile`:

- 1-й параметр указывает на область, записываемую в файл;
- 2-й параметр определяет количество записываемых байтов.

В функции `UnmapViewOfFile` единственным параметром является дескриптор отображаемого файла.

Вот, собственно, и вся теория отображаемых файлов. Материал весьма прост для программирования. Я думаю, что читатель, добравшийся до данной главы, без труда будет использовать описанные выше инструменты в своих программах. В качестве простого примера я привожу в листинге 3.5.2 простую программу, демонстрирующую возможности использования отображаемых в память файлов. Данная программа проецирует в память некоторый файл (имя файла хранится в переменной `BUF`) и осуществляет операцию `XOR` над всеми байтами файла. При этом наша программа рассчитана на работу с файлами произвольной длины (см. использование функции `GetFileSize`). Обратите также внимание на алгоритм работы с 64-битными числами.

В программе я не сбрасываю отображаемый файл на диск при помощи функции `FlushViewOfFile`, т. к. все проводимые изменения и так заносятся в файл при его закрытии.

Листинг 3.5.2. Простой пример использования файла, отображаемого в память

```
;файл MFILE.ASM
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
GENERIC_READ    equ 80000000h
GENERIC_WRITE   equ 40000000h
GEN = GENERIC_READ or GENERIC_WRITE
OPEN_EXISTING   equ 3
```

```
PAGE_READWRITE  equ 4
FILE_MAP_WRITE  equ 2
;прототипы внешних процедур
EXTERN  GetFileSize@8:NEAR
EXTERN  UnmapViewOfFile@4:NEAR
EXTERN  MapViewOfFile@20:NEAR
EXTERN  CreateFileMappingA@24:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  CreateFileA@28:NEAR
EXTERN  CloseHandle@4:NEAR
;------------------------------------------------
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\kernel32.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
      HFILE DWORD ?  ; дескриптор открытого файла
      MFILE DWORD ?  ; дескриптор отображаемого файла
      FADDR DWORD ?  ; адрес проецируемого файла
      BUF   DB    'texts.txt',0
      LB1   DD 0     ; младшая часть длины файла
      LB2   DD 0     ; старшая часть длины файла
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить параметр номером EDI
;открыть файл
      PUSH 0                 ; должен быть равен 0
      PUSH 0                 ; атрибут файла (если создаем)
      PUSH OPEN_EXISTING     ; как открывать
      PUSH 0                 ; указатель на security attr
      PUSH 0                 ; режим общего доступа
      PUSH GEN               ; режим доступа
      PUSH OFFSET BUF        ; имя файла
      CALL CreateFileA@28
      CMP  EAX,-1
      JE  _EXIT
      MOV  HFILE,EAX
;получить длину файла
      PUSH OFFSET LB2
      PUSH HFILE
      CALL GetFileSize@8
      MOV  LB1,EAX
;создать отображаемый файл
      PUSH 0   ; файл без имени
```

```
      PUSH LB1
      PUSH LB2
      PUSH PAGE_READWRITE
      PUSH 0
      PUSH HFILE
      CALL CreateFileMappingA@24
      MOV  MFILE,EAX
;спроецировать файл в память
      PUSH 0
      PUSH 0
      PUSH 0
      PUSH FILE_MAP_WRITE
      PUSH MFILE
      CALL MapViewOfFile@20
      MOV  FADDR,EAX
;выполняем действия над файлом
      CLC
      MOV  EBX,FADDR
      MOV  EDX,LB2
      MOV  EAX,LB1
LL:
      SUB  EAX,1
      JC   LL1
LL3:
      XOR  BYTE PTR [EBX],10
      INC  EBX
      JMP  LL
LL1:
      SBB  EDX,0
      JNC  LL3
;сделать указатель недействительным
      PUSH FADDR
      CALL UnmapViewOfFile@4
;закрыть отображаемый файл
      PUSH MFILE
      CALL CloseHandle@4
;закрыть файл
      PUSH HFILE
      CALL CloseHandle@4
;конец работы программы
_EXIT:
      PUSH 0
      CALL ExitProcess@4
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.5.2:

```
ml /c /coff  mfile.asm
link /subsystem:console mfile.obj
```

Вопрос: можно ли контролировать ввод информации в поле редактирования?

Да, в *главе 2.5* мы видели, как можно корректно контролировать ввод данных при помощи горячих клавиш. Однако с помощью горячих клавиш легко блокировать приход в поле редактирования некоторых символов. Речь же в некоторых случаях может идти о том, чтобы преобразовывать вводимую информацию, что называется, "на лету". Таким механизмом может быть использование подклассов. Для демонстрации его мы возьмем программу из листинга 1.3.2, видоизменим ее так, чтобы можно было контролировать ввод информации.

На самом деле в этом механизме нет ничего нового. Еще в операционной системе MS-DOS можно было перехватывать прерывание и встраивать в его обработку свою процедуру. В результате при вызове данного прерывания вызывалась вначале ваша процедура, а потом уже та, которая была установлена ранее. Впрочем, можно было поступать и по-другому: вначале вызывается процедура, существовавшая ранее, а потом, в последнюю очередь, вызывается ваша процедура (см. [1]). Поскольку перехватывать прерывание можно было многократно, в результате могла образоваться целая цепочка процедур, выполняющихся одна за другой. Мне уже приходилось сравнивать вызов процедур окна с вызовом прерываний. Это очень похоже, не правда ли? С точки зрения объектного программирования это является не чем иным, как созданием класса-родителя.

В основе рассматриваемого механизма лежит использование функции `SetWindowLong`, которая может менять атрибуты уже созданного окна. Вот параметры этой функции:

- 1-й параметр — дескриптор окна, которое было создано текущим процессом;
- 2-й параметр — величина, определяющая, какой атрибут следует изменить. Вообще говоря, это всего лишь смещение в некоторой области памяти. Нас будет интересовать только величина, определяющая адрес процедуры окна. Эта величина определяется константой `DWL_DLGPROC = 4` для диалогового окна и `GWL_WNDPROC = -4` для обычного окна. Отсюда, таким образом, следует, что данное действие можно осуществлять не только над простыми, но и над диалоговыми окнами;
- 3-й параметр — новое значение атрибута окна.

Следует также отметить, что данная функция возвращает старое значение атрибута.

Здесь я должен еще обратить внимание читателя на то, что все элементы, создаваемые в окне, также в свою очередь являются окнами, имеющими собственные функции окна. В повседневной практике мы довольствуемся только сообщениями, приходящими в процедуру основного окна.

Вопрос следующий: как вызвать старую процедуру окна. Обычная команда `CALL` не подходит. Нужно использовать специальную функцию `CallWindowProc`. Параметры этой функции следующие:

- 1-й параметр — адрес вызываемой процедуры;
- 2-й параметр — дескриптор окна;
- 3-й параметр — код сообщения;
- 4-й параметр — параметр `WPARAM`;
- 5-й параметр — параметр `LPARAM`.

В листинге 3.5.3 представлена программа, в которой демонстрируется пример использования подклассов. Мы перехватываем ввод в поле редактирования и при поступлении кода перевода строки (значение 13) посылаем сообщение, закрывающее основное окно и тем самым приложение.

Листинг 3.5.3. Пример использования подклассов

```
;файл edit.inc
;константы
WM_CHAR         equ 102h
WM_SETFOCUS     equ 7h
;сообщение приходит при закрытии окна
WM_DESTROY      equ 2
;сообщение приходит при создании окна
WM_CREATE       equ 1
;сообщение, если что-то происходит с элементами в окне
WM_COMMAND      equ 111h
;сообщение, позволяющее получить строку
WM_GETTEXT        equ 0Dh
;константа для функции SetWindowLong
GWL_WNDPROC       equ -4
;свойства окна
CS_VREDRAW        equ 1h
CS_HREDRAW        equ 2h
CS_GLOBALCLASS    equ 4000h
WS_TABSTOP        equ 10000h
WS_SYSMENU        equ 80000h
```

```
WS_OVERLAPPEDWINDOW equ 0+WS_TABSTOP+WS_SYSMENU
STYLE              equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
CS_HREDRAW         equ 2h
BS_DEFPUSHBUTTON   equ 1h
WS_VISIBLE         equ 10000000h
WS_CHILD           equ 40000000h
WS_BORDER          equ 800000h
STYLBTN            equ WS_CHILD+BS_DEFPUSHBUTTON+WS_VISIBLE+WS_TABSTOP
STYLEDT            equ WS_CHILD+WS_VISIBLE+WS_BORDER+WS_TABSTOP
;идентификатор стандартной пиктограммы
IDI_APPLICATION    equ 32512
;идентификатор курсора
IDC_ARROW          equ 32512
;режим показа окна - нормальный
SW_SHOWNORMAL      equ 1
;прототипы внешних процедур
EXTERN  CallWindowProcA@20:NEAR
EXTERN  SetWindowLongA@12:NEAR
EXTERN  SetFocus@4:NEAR
EXTERN  SendMessageA@16:NEAR
EXTERN  MessageBoxA@16:NEAR
EXTERN  CreateWindowExA@48:NEAR
EXTERN  DefWindowProcA@16:NEAR
EXTERN  DispatchMessageA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetMessageA@16:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  LoadCursorA@8:NEAR
EXTERN  LoadIconA@8:NEAR
EXTERN  PostQuitMessage@4:NEAR
EXTERN  RegisterClassA@4:NEAR
EXTERN  ShowWindow@8:NEAR
EXTERN  TranslateMessage@4:NEAR
EXTERN  UpdateWindow@4:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;----структура класса окон
WNDCLASS STRUC
      CLSSTYLE   DD ?
```

```
      CLWNDPROC  DD ?
      CLSCBCLSEX DD ?
      CLSCBWNDEX DD ?
      CLSHINST   DD ?
      CLSHICON   DD ?
      CLSHCURSOR DD ?
      CLBKGROUND DD ?
      CLMENNAME  DD ?
      CLNAME     DD ?
WNDCLASS ENDS
;файл edit.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include editn.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      NEWHWND    DD 0
      MSG        MSGSTRUCT <?>
      WC         WNDCLASS  <?>
      HINST      DD 0 ;дескриптор приложения
      TITLENAME  DB 'Контроль окна редактирования',0
      CLASSNAME  DB 'CLASS32',0
      CPBUT      DB 'Выход',0 ;выход
      CPEDT      DB ' ',0
      CLSBUTN    DB 'BUTTON',0
      CLSEDIT    DB 'EDIT',0
      HWNDBTN    DWORD 0
      HWNDEDT    DWORD 0
      CAP        BYTE 'Сообщение',0
      MES        BYTE 'Конец работы программы',0
      TEXT       DB 100 DUP(0)
      OLDWND     DD 0
      CHAR       DD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
```

```
REG_CLASS:
;заполнить структуру окна
; стиль
      MOV  WC.CLSSTYLE,STYLE
;процедура обработки сообщений
      MOV  WC.CLWNDPROC, OFFSET WNDPROC
      MOV  WC.CLSCBCLSEX, 0
      MOV  WC.CLSCBWNDEX, 0
      MOV  EAX, HINST
      MOV  WC.CLSHINST, EAX
;----------пиктограмма окна
      PUSH  IDI_APPLICATION
      PUSH  0
      CALL  LoadIconA@8
      MOV   WC.CLSHICON, EAX
;----------курсор окна
      PUSH  IDC_ARROW
      PUSH  0
      CALL  LoadCursorA@8
      MOV   WC.CLSHCURSOR, EAX
;----------
      MOV  [WC.CLBKGROUND], 17 ;цвет окна
      MOV   DWORD PTR WC.CLMENNAME,0
      MOV   DWORD PTR WC.CLNAME,OFFSET CLASSNAME
      PUSH  OFFSET WC
      CALL  RegisterClassA@4
;создать окно зарегистрированного класса
      PUSH  0
      PUSH  [HINST]
      PUSH  0
      PUSH  0
      PUSH  150      ; DY - высота окна
      PUSH  400      ; DX - ширина окна
      PUSH  100      ; Y-координата левого верхнего угла
      PUSH  100      ; X-координата левого верхнего угла
      PUSH  WS_OVERLAPPEDWINDOW
      PUSH  OFFSET TITLENAME   ; имя окна
      PUSH  OFFSET CLASSNAME   ; имя класса
      PUSH  0
      CALL  CreateWindowExA@48
;проверка на ошибку
      CMP  EAX,0
      JZ  _ERR
      MOV  NEWHWND, EAX        ; дескриптор окна
;-----------------------------------
      PUSH  SW_SHOWNORMAL
```

```
      PUSH  NEWHWND
      CALL  ShowWindow@8       ; показать созданное окно
;-----------------------------------
      PUSH  NEWHWND
      CALL  UpdateWindow@4     ; команда перерисовать видимую
                               ; часть окна, сообщение WM_PAINT
;цикл обработки сообщений
MSG_LOOP:
      PUSH  0
      PUSH  0
      PUSH  0
      PUSH  OFFSET MSG
      CALL  GetMessageA@16
      CMP   AX, 0
      JE    END_LOOP
      PUSH  OFFSET MSG
      CALL  TranslateMessage@4
      PUSH  OFFSET MSG
      CALL  DispatchMessageA@4
      JMP   MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
      PUSH  MSG.MSWPARAM
      CALL  ExitProcess@4
_ERR:
      JMP  END_LOOP
;--------------------------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
      CMP  DWORD PTR [EBP+0CH],WM_DESTROY
      JE   WMDESTROY
      CMP  DWORD PTR [EBP+0CH],WM_CREATE
      JE   WMCREATE
      CMP  DWORD PTR [EBP+0CH],WM_COMMAND
      JE   WMCOMMND
      JMP  DEFWNDPROC
```

```
WMCOMMND:
      MOV  EAX,HWNDBTN
      CMP  DWORD PTR [EBP+14H],EAX
      JNE  NODESTROY
      JMP  WMDESTROY
NODESTROY:
      MOV  EAX, 0
      JMP  FINISH
WMCREATE:
;создать окно-кнопку
      PUSH  0
      PUSH  HINST
      PUSH  0
      PUSH  DWORD PTR [EBP+08H]
      PUSH  20      ; DY
      PUSH  60      ; DX
      PUSH  10      ; Y
      PUSH  10      ; X
      PUSH  STYLBTN
      PUSH  OFFSET CPBUT     ; имя окна
      PUSH  OFFSET CLSBUTN   ; имя класса
      PUSH  0
      CALL  CreateWindowExA@48
      MOV   HWNDBTN,EAX      ; запомнить дескриптор кнопки
;создать окно редактирования
      PUSH  0
      PUSH  HINST
      PUSH  0
      PUSH  DWORD PTR [EBP+08H]
      PUSH  20      ; DY
      PUSH  350     ; DX
      PUSH  50      ; Y
      PUSH  10      ; X
      PUSH  STYLEDT
      PUSH  OFFSET CPEDT     ; имя окна
      PUSH  OFFSET CLSEDIT   ; имя класса
      PUSH  0
      CALL  CreateWindowExA@48
      MOV   HWNDEDT,EAX
;установить фокус в поле редактирования
      PUSH  HWNDEDT
      CALL  SetFocus@4
;установить собственную процедуру обработки
;для поля редактирования
      PUSH  OFFSET WNDEDIT
      PUSH  GWL_WNDPROC
```

```
      PUSH  HWNDEDT
      CALL  SetWindowLongA@12
      MOV   OLDWND,EAX
      MOV   EAX, 0
      JMP   FINISH
DEFWNDPROC:
      PUSH  DWORD PTR [EBP+14H]
      PUSH  DWORD PTR [EBP+10H]
      PUSH  DWORD PTR [EBP+0CH]
      PUSH  DWORD PTR [EBP+08H]
      CALL  DefWindowProcA@16
      JMP   FINISH
WMDESTROY:
;получить отредактированную строку
      PUSH  OFFSET TEXT
      PUSH  150
      PUSH  WM_GETTEXT
      PUSH  HWNDEDT
      CALL  SendMessageA@16
;показать эту строку
      PUSH  0
      PUSH  OFFSET CAP
      PUSH  OFFSET TEXT
      PUSH  DWORD PTR [EBP+08H] ;дескриптор окна
      CALL  MessageBoxA@16
;на выход
      PUSH  0
      CALL  PostQuitMessage@4 ;сообщение WM_QUIT
      MOV   EAX, 0
FINISH:
      POP   EDI
      POP   ESI
      POP   EBX
      POP   EBP
      RET   16
WNDPROC  ENDP
;-----------------------------------------------
;новая процедура обработки сообщений окну редактирования
WNDEDIT  PROC
      PUSH EBP
      MOV  EBP,ESP
      MOV  EAX,DWORD PTR [EBP+10H]
      MOV  CHAR,EAX
      CMP  DWORD PTR [EBP+0CH],WM_CHAR
      JNE _OLD
;проверка вводимого символа
```

```
      CMP  AL,13
      JNE _OLD
;послать сообщение о закрытии основного окна
      PUSH 0
      PUSH 0
      PUSH WM_DESTROY
      PUSH NEWHWND
      CALL SendMessageA@16
_OLD:
;вызвать старую процедуру
      PUSH DWORD PTR [EBP+014H]
      PUSH DWORD PTR [EBP+10H]
      PUSH DWORD PTR [EBP+0CH]
      PUSH DWORD PTR [EBP+8H]
      PUSH OLDWND
      CALL CallWindowProcA@20
FIN:
      POP  EBP
      RET  16
WNDEDIT  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.5.3:

```
ml /c /coff  editn.asm
link /subsystem:windows editn.obj
```

Разбирая программу из листинга 3.5.3, обратите внимание, что представленная схема позволяет проделывать с полем редактирования практически любые трюки. К примеру, вы можете заблокировать любой символ, послав вместо него код 0, или вместо одного символа подставить другой и, наконец, выдавать список возможных значений для уже введенной части слова или фразы.

Вопрос: как можно осуществлять обмен информацией между различными процессами?

Мы уже говорили о различных способах синхронизации потоков, о разделяемой памяти, о файлах, проецируемых в память. Все эти средства можно использовать для обмена данными между процессами. Но есть еще один интересный подход, реализованный в Windows, — это *анонимные (неименованные) каналы* (pipes)[2]. Этот подход наиболее эффективен для обмена информацией с консольным процессом, порождаемым данным приложением.

[2] О каналах, в том числе именованных, *см. также в главе 2.8*.

Представьте себе, что вам необходимо, чтобы запускаемый вами из приложения консольный процесс (например, какой-нибудь строковый компилятор) выводил информацию не в консоль, а в окно редактирования основного процесса. Пример такого приложения представлен в листинге 3.5.4.

Идея использования каналов очень проста. Канал как труба: с одной стороны в него втекает информация, а с другой вытекает. Создавая процесс, можно передать ему в качестве дескриптора ввода или вывода соответствующий дескриптор канала. После этого можно обмениваться информацией между двумя процессами при помощи уже известных вам функций `WriteFile` и `ReadFile`.

Листинг 3.5.4. Пример взаимодействия с консольным процессом через анонимный канал

```
//файл pipe.rc
//определение констант
#define WS_SYSMENU    0x00080000L
#define WS_VISIBLE    0x10000000L
#define WS_TABSTOP    0x00010000L
#define WS_VSCROLL    0x00200000L
#define DS_3DLOOK     0x0004L
#define ES_LEFT       0x0000L
#define WS_CHILD      0x40000000L
#define WS_BORDER     0x00800000L
#define ES_MULTILINE  0x0004L
#define WS_VSCROLL    0x00200000L
#define WS_HSCROLL    0x00100000L

MENUP MENU
{
 POPUP "&Запуск программы"
 {
 MENUITEM "&Запустить", 200
 MENUITEM "Выход из &программы", 300
 }
}

//определение диалогового окна
DIAL1 DIALOG 0, 0, 200, 140
STYLE WS_SYSMENU | DS_3DLOOK
CAPTION "Пример использования PIPE"
FONT 8, "Arial"
{
 CONTROL "", 101, "edit", ES_LEFT | ES_MULTILINE
| WS_VISIBLE | WS_BORDER | WS_VSCROLL
```

```
| WS_HSCROLL , 24, 20, 128, 70
}
;файл pipe.inc
;константы
SW_HIDE                equ 0
SW_SHOWNORMAL          equ 1
STARTF_USESHOWWINDOW  equ 1h
STARTF_USESTDHANDLES  equ 100h
STARTF_ADD = STARTF_USESHOWWINDOW or STARTF_USESTDHANDLES
;сообщение приходит при закрытии окна
WM_CLOSE               equ 10h
WM_INITDIALOG          equ 110h
WM_COMMAND             equ 111h
EM_REPLACESEL          equ 0C2h
;прототипы внешних процедур
EXTERN ReadFile@20:NEAR
EXTERN CloseHandle@4:NEAR
EXTERN CreatePipe@16:NEAR
EXTERN SetMenu@8:NEAR
EXTERN LoadMenuA@8:NEAR
EXTERN CreateProcessA@40:NEAR
EXTERN ExitProcess@4:NEAR
EXTERN GetModuleHandleA@4:NEAR
EXTERN DialogBoxParamA@20:NEAR
EXTERN EndDialog@8:NEAR
EXTERN SendDlgItemMessageA@20:NEAR
EXTERN GetStartupInfoA@4:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;структура для CreateProcess
STARTUP STRUC
      cb                DD 0
      lpReserved        DD 0
      lpDesktop         DD 0
      lpTitle           DD 0
      dwX               DD 0
      dwY               DD 0
      dwXSize           DD 0
```

```
      dwYSize          DD 0
      dwXCountChars    DD 0
      dwYCountChars    DD 0
      dwFillAttribute  DD 0
      dwFlags          DD 0
      wShowWindow      DW 0
      cbReserved2      DW 0
      lpReserved2      DD 0
      hStdInput        DD 0
      hStdOutput       DD 0
      hStdError        DD 0
STARTUP ENDS
;структура - информация о процессе
PROCINF STRUC
      hProcess DD ?
      hThread  DD ?
      Idproc   DD ?
      idThr    DD ?
PROCINF ENDS

SECURITY_ATTRIBUTES STRUCT
  nLength               DWORD  ?
  lpSecurityDescriptor  DWORD  ?
  bInheritHandle        DWORD  ?
SECURITY_ATTRIBUTES ENDS
;файл pipe.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include pipe.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
      STRUP  STARTUP   <?>
      INF    PROCINF   <?>
      MSG    MSGSTRUCT <?>
      HINST  DD 0    ; дескриптор приложения
      PA     DB "DIAL1",0
      CMD    DB "c:\masm32\bin\link.exe",0
      PMENU  DB "MENUP",0
      HW     DD ?
      HR     DD ?
      BUFER  DB 8000 DUP(0)
```

```
      BYT    DD ?
      SAT    SECURITY_ATTRIBUTES <>
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
;--------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  [HINST]
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
;сообщение об ошибке
KOL:
;---------------------------------
      PUSH  0
      CALL  ExitProcess@4
;---------------------------------
;процедура окна
;расположение параметров в стеке
; [BP+014H]  ;LPARAM
; [BP+10H]   ;WAPARAM
; [BP+0CH]   ;MES
; [BP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;----------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
L3:
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP  FINISH
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
```

```
        JNE  L2
;загрузить меню
        PUSH OFFSET PMENU
        PUSH HINST
        CALL LoadMenuA@8
;установить меню
        PUSH EAX
        PUSH DWORD PTR [EBP+08H]
        CALL SetMenu@8
        JMP  FINISH
L2:
        CMP  DWORD PTR [EBP+0CH],WM_COMMAND
        JNE  FINISH
        CMP  WORD PTR [EBP+10H],300
        JE   L3
        CMP  WORD PTR [EBP+10H],200
        JNE  FINISH
;здесь запуск
;в начале PIPE
        MOV  SAT.nLength,sizeof SECURITY_ATTRIBUTES
        MOV  SAT.lpSecurityDescriptor,0
        MOV  SAT.bInheritHandle,1
        PUSH 0
        PUSH OFFSET SAT
        PUSH OFFSET HW
        PUSH OFFSET HR
        CALL CreatePipe@16
;задать структуру STARTUP
        MOV  STRUP.cb,sizeof STRUP
        PUSH OFFSET STRUP
        CALL GetStartupInfoA@4
        MOV  EAX,HW
        MOV  STRUP.dwFlags,STARTF_ADD
        MOV  STRUP.wShowWindow,SW_HIDE ;окно процесса невидимо
        MOV  STRUP.hStdOutput,EAX
        MOV  STRUP.hStdError,EAX
;здесь запуск консольного приложения
        PUSH OFFSET INF
        PUSH OFFSET STRUP
        PUSH 0
        PUSH 0
        PUSH 0
        PUSH 1 ;наследует дескрипторы
        PUSH 0
        PUSH 0
        PUSH OFFSET CMD
```

```
      PUSH 0 ;OFFSET CMD
      CALL CreateProcessA@40
      CMP  EAX,0
      JZ   FINISH
;закрыть дескриптор на запись
      PUSH HW
      CALL CloseHandle@4
;здесь чтение информации
LL:
      PUSH 0
      PUSH OFFSET BYT
      PUSH 8000
      PUSH OFFSET BUFER
      PUSH HR
      CALL ReadFile@20
;занести информацию в поле редактирования
      PUSH OFFSET BUFER
      PUSH 0
      PUSH EM_REPLACESEL
      PUSH 101
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
;проверка — не закончилось ли чтение
      CMP BYT,0
      JNZ LL
;закрыть дескриптор на чтение
      PUSH HR
      CALL CloseHandle@4
FINISH:
      MOV EAX,0
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      RET 16
WNDPROC  ENDP
_TEXT ENDS
END START
```

Трансляция программы, представленной в листинге 3.5.4:

```
ml /c /coff  pipe.asm
rc pipe.rc
link /subsystem:windows pipe.obj pipe.res
```

На рис. 3.5.2 представлено окно программы.

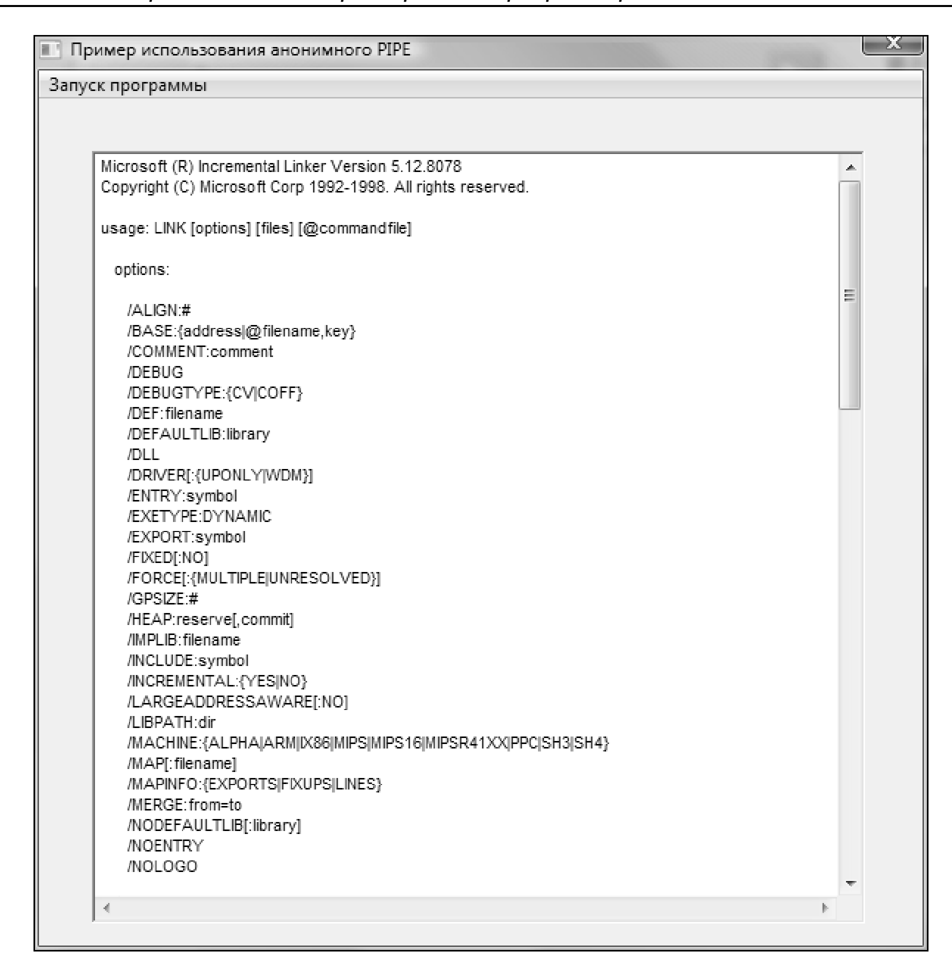


Рис. 3.5.2. Окно программы, осуществляющей перехват вывода на консоль

Прокомментирую программу из листинга 3.5.4.

- В нашей программе запуск приложения почти не отличается от запуска программы WINWORD.EXE в *главе 3.2*. Но здесь запускается консольное приложение. Отмечу новое для вас. Обратите внимание, что управляющий элемент `EditBox` играет здесь несколько необычную роль. По сути, этот элемент играет роль консоли вывода. Для этого мы указали свойство `ES_MULTILINE`, что дает возможность помещать в окно целый текст, который отправляется в окно при помощи сообщения `EM_REPLACESEL`. Для чтения информации мы используем довольно большой буфер. В принципе, как и в случае с файлами, можно читать несколькими порциями, проверяя количество прочитанных байтов.

- Обратите также внимание, как перехватывается консольный вывод:

  ```
  MOV STRUP.hStdOutput,EAX
  MOV STRUP.hStdError,EAX
  ```

  При этом в регистре `EAX` находится дескриптор вывода, который мы получили при вызове функции `CreatePipe` — создание неименованного канала.
- Обратим также внимание на вызов функции `GetStartupInfo`, которая заполняет структуру `STARTUP` данными, определенными при создании текущего процесса. В нашем случае вызов этой функции можно было бы опустить.

Вопрос: можно ли запретить многократный запуск одного и того же приложения?

Наиболее часто употребляемым средством для контроля повторного запуска приложения является создание объекта mutex. Этот объект как раз и предназначен для того, чтобы координировать разные процессы. Создается данный объект при помощи функции `CreateMutex`. Рассмотрим параметры этой функции:

- 1-й параметр — указатель на структуру, определяющую атрибут безопасности. Обычно полагают равным `NULL` (0);
- 2-й параметр — флаг. В случае ненулевого значения процесс требует немедленного владения объектом (!);
- 3-й параметр — указатель на имя объекта.

При запуске программы она создает объект mutex. Второй параметр должен быть ненулевым. При вторичном запуске программы:

- будет сгенерирована ошибка, если запуск осуществлялся от имени пользователя, не имеющего право на доступ к данному объекту;
- новый объект не будет создан, но будет открыт уже созданный объект. В этом случае требуется вызов функции `GetLastError`, и проверки, не возвратила ли она значение `ERROR_ALREADY_EXISTS equ 183`. Это и будет признаком того, что запущен еще один экземпляр программы.

В листинге 3.5.5 представлена простая консольная программа, демонстрирующая использование объекта mutex для проверки вторичного запуска.

К тому же результату можно прийти, используя семафор или файл, отображаемый в память. В данном случае все достаточно тривиально.

Еще один подход основан на разделяемой памяти. Определим область разделяемой памяти и там — переменную. При запуске приложение проверяет значение переменной, и если она равна нулю, то помещает туда единицу. Если переменная уже равна единице, то выход (или действия, предусмотренные на этот случай).

Листинг 3.5.5. Простой пример использования объекта mutex для распознавания вторичного запуска программы

```
;программа mutex.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_INPUT_HANDLE      equ -10
ERROR_ALREADY_EXISTS  equ 183
;атрибуты цветов
;прототипы внешних процедур
EXTERN  GetLastError@0:NEAR
EXTERN  CloseHandle@4:NEAR
EXTERN  CreateMutexA@12:NEAR
EXTERN  GetStdHandle@4:NEAR
EXTERN  SetConsoleTitleA@4:NEAR
EXTERN  FreeConsole@0:NEAR
EXTERN  AllocConsole@0:NEAR
EXTERN  ReadConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      HANDLE DD ?
      HANDL1 DD ?
      BUF    DB 200 dup(?)
      LENS   DD ?
      NMUTEX DB "mutex133",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;создать объект mutex
      PUSH OFFSET nmutex
      PUSH 1
      PUSH 0
      CALL CreateMutexA@12
;проверка на ошибку
      CMP  EAX,0
      JZ  _EXIT
      MOV  HANDLE,EAX
;проверим, нет ли уже созданного объекта с таким именем
```

```
      CALL GetLastError@0
      CMP  EAX,ERROR_ALREADY_EXISTS
      JZ   _EXIT
;образовать консоль
;вначале освободить уже существующую
      CALL FreeConsole@0
      CALL AllocConsole@0
;получить HANDL1 ввода
      PUSH STD_INPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL1,EAX
;ждать ввод строки
      PUSH 0
      PUSH OFFSET LENS
      PUSH 200
      PUSH OFFSET BUF
      PUSH HANDL1
      CALL ReadConsoleA@20
      PUSH HANDLE
      CALL CloseHandle@4
      CALL FreeConsole@0
_EXIT:
      CALL ExitProcess@4
_TEXT ENDS
END START
```

Трансляция программы, представленной в листинге 3.5.5:

```
ml /c /coff  mutex.asm
link /subsystem:console mutex.obj
```

Вопрос: имеет ли операционная система Windows средства, упрощающие операции над группами файлов и каталогами?

Да, имеется функция `SHFileOperation`, которая умеет выполнять копирование, перенос, переименование или удаление файловых объектов (т. е. файлов и каталогов, в том числе и вложенных). Функция располагается в системной динамической библиотеке Shell32.dll. Данная функция имеет всего один параметр — указатель на структуру, которая и определяет, какую операцию следует произвести, над чем и как. Вот эта структура:

```
SH struct
      hwnd                  DD ?
      wFunc                 DD ?
      pFrom                 DD ?
      pTo                   DD ?
      fFlags                DD ?
```

```
    fAnyOperationsAborted  DD ?
    hNameMappings          DD ?
    lpszProgressTitle      DD ?
SH ENDS
```

Рассмотрим значение этих полей:

- hwnd — дескриптор окна, куда будет выводиться статус операции;
- wFunc — код операции. Может принимать следующие значения:
  - FO_COPY equ 2h — копирование файлов, которые определяются параметром pFrom, в каталог, определяемый параметром pTo;
  - FO_DELETE equ 3h — удаление файлов, определяемых параметром pFrom;
  - FO_MOVE equ 1h — перемещение файлов, которые определяются параметром pFrom, в каталог, определяемый параметром pTo;
  - FO_RENAME equ 4h — переименование файла, определяемого параметром pFrom. Для переименования сразу нескольких файлов следует использовать операцию FO_MOVE;
- pFrom — название файла, каталога или группы файлов или каталогов, над которыми будет производиться операция. Если объектов несколько, то их имена отделяются друг от друга символами с кодом 0. Можно выделять списки, которые разделяются двумя нулевыми символами;
- pTo — имя или группа имен объектов, которые должны быть получены в результате копирования. Здесь действует то же правило: элементы списка отделяются кодом 0. В конце списка два нулевых символа;
- fFlags — флаг, определяет характер операции и образуется комбинацией следующих констант:
  - FOF_ALLOWUNDO — сохранить, если возможно, информацию для возвращения в исходное состояние;
  - FOF_CONFIRMMOUSE — данное значение не реализовано;
  - FOF_FILESONLY — выполнять только над файлами, если определен шаблон;
  - FOF_MULTIDESTFILES — указывает, что pTo содержит несколько результирующих файлов или каталогов. Например, можно копировать сразу в несколько каталогов. Если pFrom состоит из нескольких файлов, то каждый файл будет копироваться в свой каталог;
  - FOF_NOCONFIRMATION — отвечать утвердительно на все запросы;
  - FOF_NOCONFIRMMKDIR — не подтверждать создание каталога, если это требуется;

- FOF_NO_CONNECTED_ELEMENTS — не копировать связанные файлы. Имеются в виду файлы, связанные с HTML-документом. Связь определяется по имени HTML-документа. Например, если документ имеет имя first.htm, то связанные файлы должны находиться в подкаталоге first.files в том же каталоге, что и HTML-документ;
- FOF_NOCOPYSECURITYATTRIBS — не копировать атрибуты безопасности;
- FOF_NOERRORUI — не выводить какие-либо элементы управления в случае возникновения ошибки;
- FOF_NORECURSION — не выполнять рекурсивные операции над каталогами;
- FOF_NORECURSEREPARSE — рассматривать точки повторной обработки как объекта, а не как контейнеры;
- FOF_RENAMEONCOLLISION — давать файлам новые имена, если файлы с такими именами уже существуют;
- FOF_SILENT — не показывать окно-статус;
- FOF_SIMPLEPROGRESS — показывать окно-статус, но не показывать имена файлов;
- FOF_WANTMAPPINGHANDLE — заполнять отображаемый файл *(см. далее)*;
- FOF_WANTNUKEWARNING — посылать предупреждение, если файл во время удаления был поврежден;

- fAnyOperationsAborted — переменная, по значению которой после операции можно определить, была ли прервана операция (не равна 0) или нет (0);
- hNameMappings — дескриптор отображаемого в памяти файла, содержащего массив, который состоит из новых и старых имен файлов, участвующих в операции;
- lpszProgressTitle — указывает на строку-заголовок для диалогового окна-статуса.

Кроме описанной функции, есть еще целая группа функций, начинающихся с префикса SH. Среди них особенно полезна функция SHGetDesktopFolder, осуществляющая вывод диалогового окна для выбора нужной папки каталога.

В листинге 3.5.6 представлена простая консольная программа копирования. С ее помощью можно копировать как отдельные файлы, так и каталоги. Программа не позволяет создавать списки копирования. Попробуйте добавить в нее эту возможность.

Листинг 3.5.6. Простая программа копирования

```
;программа copy.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_INPUT_HANDLE      equ -10
STD_OUTPUT_HANDLE     equ -11
ERROR_ALREADY_EXISTS  equ 183
FO_COPY               equ 2h
;атрибуты цветов
;прототипы внешних процедур
EXTERN  SHFileOperationA@4:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  CharToOemA@8:NEAR
EXTERN  GetLastError@0:NEAR
EXTERN  CloseHandle@4:NEAR
EXTERN  GetStdHandle@4:NEAR
EXTERN  FreeConsole@0:NEAR
EXTERN  AllocConsole@0:NEAR
EXTERN  ReadConsoleA@20:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\shell32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
SHFILEOPSTRUCTA STRUCT
      hwnd                  DD  ?
      wFunc                 DD  ?
      pFrom                 DD  ?
      pTo                   DD  ?
      fFlags                DW  ?
      fAnyOperationsAborted DD  ?
      hNameMappings         DD  ?
      lpszProgressTitle     DD  ?
      VAL                   DD  ?
SHFILEOPSTRUCTA ENDS
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      HANDLE1 DD ?
      HANDLE2 DD ?
      BUF1    DB 260 dup(0)
```

```
      BUF2    DB 260 dup(0)
      LENS    DD ?
      SH      SHFILEOPSTRUCTA <0>
      TEXT1   DB 'Копировать из:',0
      TEXT2   DB 'В:',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;перекодировать строки
      PUSH OFFSET TEXT1
      PUSH OFFSET TEXT1
      CALL CharToOemA@8
      PUSH OFFSET TEXT2
      PUSH OFFSET TEXT2
      CALL CharToOemA@8
;образовать консоль
;вначале освободить уже существующую
      CALL FreeConsole@0
      CALL AllocConsole@0
;получить HANDLE ввода
      PUSH STD_INPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDLE1,EAX
;получить HANDLE вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDLE2,EAX
;ждать ввод строки
;ввести данные
;определить длину строки
      PUSH OFFSET TEXT1
      CALL lstrlenA@4
      PUSH 0
      PUSH OFFSET LENS
      PUSH EAX
      PUSH OFFSET TEXT1
      PUSH HANDLE2
      CALL WriteConsoleA@20
      PUSH 0
      PUSH OFFSET LENS
      PUSH 260
      PUSH OFFSET BUF1
      PUSH HANDLE1
      CALL ReadConsoleA@20
```

```
      LEA  EBX,BUF1
      MOV  EAX,LENS
      MOV  WORD PTR [EBX+EAX-2],0
;определить длину строки
      PUSH OFFSET TEXT2
      CALL lstrlenA@4
      PUSH 0
      PUSH OFFSET LENS
      PUSH EAX
      PUSH OFFSET TEXT2
      PUSH HANDLE2
      CALL WriteConsoleA@20
      PUSH 0
      PUSH OFFSET LENS
      PUSH 260
      PUSH OFFSET BUF2
      PUSH HANDLE1
      CALL ReadConsoleA@20
      LEA  EBX,BUF2
      MOV  EAX,LENS
      MOV  WORD PTR [EBX+EAX-2],0
;выполнить копирование
;в начале заполним структуру
      MOV  SH.hwnd,0
      MOV  SH.wFunc,FO_COPY ;операция копирования
      MOV  SH.pFrom,OFFSET BUF1
      MOV  SH.pTo,OFFSET BUF2
      MOV  SH.fFlags,0
;-------------------------------
      PUSH OFFSET SH
      CALL SHFileOperationA@4
;закрыть все и выйти
      PUSH HANDLE1
      CALL CloseHandle@4
      PUSH HANDLE2
      CALL CloseHandle@4
      CALL FreeConsole@0
_EXIT:
      CALL ExitProcess@4
_TEXT ENDS
END START
```

Трансляция программы, представленной в листинге 3.5.6:

```
ml /c /coff  copy.asm
link /subsystem:console copy.obj
```

Следует отметить один важный момент. При вводе с консоли с помощью функции `ReadConsole` в конец строки добавляются служебные коды перевода строки. Мы учитываем это в строках программы:

```
LEA EBX,BUF2
MOV EAX,LENS
MOV WORD PTR [EBX+EAX-2],0
```

Вопрос: как получить список созданных в системе окон?

В основу метода положена функция `EnumWindows` (т. е. пересчитать окна). Параметры этой функции:

- 1-й параметр — адрес процедуры, которая вызывается автоматически, если будет найдено окно;
- 2-й параметр — произвольное значение, которое будет передаваться в процедуру.

Сама вызываемая процедура получает два параметра: дескриптор найденного окна и определенный выше параметр. По известному дескриптору с помощью функции `GetClassName` можно определить название класса данного окна. Параметры этой функции:

- 1-й параметр — дескриптор окна;
- 2-й параметр — указатель на буфер, куда будет помещено имя класса окна;
- 3-й параметр — длина буфера (количество символов).

Если функция выполнилась успешно, то возвращается количество символов в имени класса окна, в противном случае возвращается 0.

В листинге 3.5.7 представлен пример программы, выдающей список всех окон системы. Обратите внимание на использование функции `GetWindowText`, которая определяет текст заголовка окна. Не пугайтесь, что в списке будут представлены и те окна, которые мы не видим. Окна, как известно, могут быть скрытыми. Заметьте, что в список попадают и консольные приложения.

На рис. 3.5.3 представлен результат работы программы. В списке через `;` перечислены дескриптор окна, заголовок окна и имя класса окна.

Листинг 3.5.7. Программа поиска окон, созданных в системе

```
//файл windows.rc
//определение констант
#define WS_SYSMENU     0x00080000L
#define WS_MINIMIZEBOX 0x00020000L
#define WS_VISIBLE     0x10000000L
#define WS_TABSTOP     0x00010000L
```

```
#define WS_VSCROLL     0x00200000L
#define WS_HSCROLL     0x00100000L
#define DS_3DLOOK      0x0004L
#define LBS_NOTIFY     0x0001L
#define LBS_SORT       0x0002L
#define BS_PUSHBUTTON  0x00000000L
#define BS_CENTER      0x00000300L
#define WS_CHILD       0x40000000L

//идентификаторы
#define LIST1      101

//определение диалогового окна
DIAL1 DIALOG 0, 0, 350, 295
STYLE WS_SYSMENU | WS_MINIMIZEBOX |
DS_3DLOOK
CAPTION "Поиск окон"
FONT 9,  "Arial"
{
 CONTROL "ListBox1",LIST1,"listbox", WS_VISIBLE |
 WS_TABSTOP | WS_VSCROLL | WS_HSCROLL |
 LBS_NOTIFY,
 16, 16, 320, 250
//кнопка, идентификатор 102
 CONTROL "Обновить", 102, "button", BS_PUSHBUTTON
 | BS_CENTER | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
 295,270,40,15

}
;файл windows.inc
;константы
;сообщение приходит при закрытии окна
WM_CLOSE        equ 10h
WM_INITDIALOG   equ 110h
WM_COMMAND      equ 111h
LB_ADDSTRING    equ 180h
LB_RESETCONTENT equ 184h
WM_LBUTTONDOWN  equ 201h
LB_RESETCONTENT equ 184h
;прототипы внешних процедур
EXTERN GetClassNameA@12:NEAR
EXTERN wsprintfA:NEAR
EXTERN GetWindowTextA@12:NEAR
EXTERN EnumWindows@8:NEAR
EXTERN lstrcatA@8:NEAR
EXTERN ExitProcess@4:NEAR
```

```
EXTERN GetModuleHandleA@4:NEAR
EXTERN DialogBoxParamA@20:NEAR
EXTERN EndDialog@8:NEAR
EXTERN SendDlgItemMessageA@20:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;файл windows.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include windows.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;---------------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG     MSGSTRUCT <?>
      HINST   DD 0 ;дескриптор приложения
      PA      DB "DIAL1",0
      BUFER1  DB 64 DUP(0)
      BUFER2  DB 64 DUP(0)
      BUF     DB 128 DUP(0)
      FORM    DB "%lu;%s;%s",0
      IDP     DD ?
      HWN     DD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
;---------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
```

```
      PUSH  OFFSET PA
      PUSH  HINST
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
;сообщение об ошибке
KOL:
;-----------------------------------
      PUSH  0
      CALL  ExitProcess@4
;-----------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;----------------
      CMP  DWORD PTR [EBP+0CH],WM_COMMAND
      JNE  L3
;кнопка?
      CMP  WORD PTR [EBP+10H],102
      JE   L2
L3:
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP  FINISH
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  FINISH
;запомним дескриптор окна
      MOV  EAX,DWORD PTR [EBP+08H]
      MOV  HWN,EAX
L2:
;очистить список
      PUSH 0
      PUSH 0
```

```
      PUSH LB_RESETCONTENT
      PUSH 101
      PUSH HWN
      CALL SendDlgItemMessageA@20
;вызвать функцию EnumWindows
      PUSH 1 ;неиспользуемый параметр
      PUSH OFFSET PENUM
      CALL EnumWindows@8
FINISH:
      MOV EAX,0
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      RET 16
WNDPROC  ENDP
;процедура обратного вызова при поиске окон
; [EBP+0CH]   ; параметр
; [EBP+8]     ; дескриптор окна
PENUM PROC
      PUSH EBP
      MOV  EBP,ESP
;получить заголовок окна
      PUSH 200
      PUSH OFFSET BUFER1
      PUSH DWORD PTR [EBP+8]
      CALL GetWindowTextA@12
;получить имя класса
      PUSH 200
      PUSH OFFSET BUFER2
      PUSH DWORD PTR [EBP+8]
      CALL GetClassNameA@12
;сформировать строку для списка
      PUSH OFFSET BUFER2
      PUSH OFFSET BUFER1
      PUSH DWORD PTR [EBP+8]
      PUSH OFFSET FORM
      PUSH OFFSET BUF
      CALL wsprintfA
;освобождаем стек
      ADD ESP,20
;добавить в список
      PUSH OFFSET BUF
      PUSH 0
      PUSH LB_ADDSTRING
      PUSH 101
```

```
        PUSH HWN
        CALL SendDlgItemMessageA@20
        POP  EBP
        MOV  EAX,1
        RET  8
PENUM ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.5.7:

```
ml /c /coff  windows.asm
rc windows.rc
link /subsystem:windows windows.obj windows.res
```

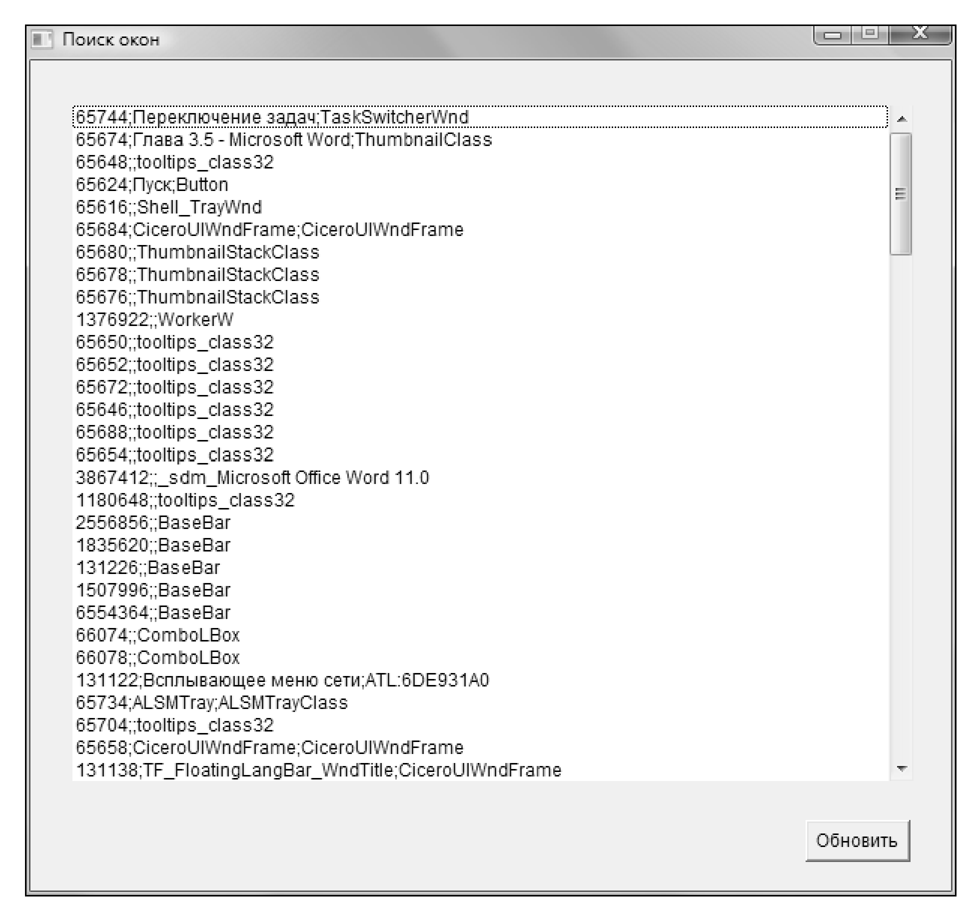


Рис. 3.5.3. Список окон, созданных в системе

Вопрос: как получить список созданных в системе процессов?

Для получения идентификаторов процессов в системе можно воспользоваться функцией `EnumProcesses`. Параметры функции:

- 1-й параметр — указатель на массив двойных слов, куда при удачном выполнении функции будут помещены идентификаторы процессов системы;
- 2-й параметр — размер массива в байтах;
- 3-й параметр — указатель на переменную, которая получит количество байтов в результирующем массиве идентификаторов.

Чтобы получить дополнительную информацию о процессе, следует его открыть функцией `OpenProcess`. Функция возвращает дескриптор процесса. Параметры функции:

- 1-й параметр — права доступа к открываемому процессу;
- 2-й параметр — наследуемость дескриптора. Если параметр равен 0, то дескриптор не наследуем;
- 3-й параметр — идентификатор процесса.

Наконец, для получения полного имени какого-либо из модулей процесса можно использовать функцию `GetModuleFileNameEx`:

- 1-й параметр — дескриптор процесса;
- 2-й параметр — дескриптор модуля, имя которого необходимо получить. Если параметр равен 0, то возвращается полное имя исполняемого файла, породившего процесс (главный модуль);
- 3-й параметр — буфер, куда будет помещено полное имя исполняемого модуля;
- 4-й параметр — количество символов, которое может уместиться в буфере.

Пример программы, которая демонстрирует механизм получения информации о процессах системы, представлен в листинге 3.5.8. На рис. 3.5.4 можно видеть окно со списком процессов.

Листинг 3.5.8. Программа поиска окон, созданных в системе

```
//файл process.rc
//определение констант
#define WS_SYSMENU     0x00080000L
#define WS_MINIMIZEBOX 0x00020000L
#define WS_VISIBLE     0x10000000L
#define WS_TABSTOP     0x00010000L
```

```
#define WS_VSCROLL     0x00200000L
#define WS_HSCROLL     0x00100000L
#define DS_3DLOOK      0x0004L
#define LBS_NOTIFY     0x0001L
#define LBS_SORT       0x0002L
#define BS_PUSHBUTTON  0x00000000L
#define BS_CENTER      0x00000300L
#define WS_CHILD       0x40000000L

//идентификаторы
#define LIST1      101

//определение диалогового окна
DIAL1 DIALOG 0, 0, 450, 295
STYLE WS_SYSMENU | WS_MINIMIZEBOX |
DS_3DLOOK
CAPTION "Поиск процессов"
FONT 8,  "Ariel"
{
 CONTROL "ListBox1",LIST1,"listbox", WS_VISIBLE |
 WS_TABSTOP | WS_VSCROLL | WS_HSCROLL |
 LBS_NOTIFY,
 16, 16, 420, 250
//кнопка, идентификатор 102
 CONTROL "Обновить", 102, "button", BS_PUSHBUTTON
 | BS_CENTER | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
 395,270,40,15

}
;файл process.inc
;константы
;сообщение приходит при закрытии окна
WM_CLOSE          equ 10h
WM_INITDIALOG     equ 110h
WM_COMMAND        equ 111h
LB_ADDSTRING      equ 180h
LB_RESETCONTENT   equ 184h
WM_LBUTTONDOWN    equ 201h
LB_RESETCONTENT   equ 184h
PROCESS_QUERY_INFORMATION equ (0400h)
PROCESS_VM_READ   equ (0010h)
;прототипы внешних процедур
EXTERN GetModuleFileNameExA@16:NEAR
EXTERN CloseHandle@4:NEAR
```

```
EXTERN OpenProcess@12:NEAR
EXTERN EnumProcesses@12:NEAR
EXTERN wsprintfA:NEAR
EXTERN lstrcatA@8:NEAR
EXTERN ExitProcess@4:NEAR
EXTERN GetModuleHandleA@4:NEAR
EXTERN DialogBoxParamA@20:NEAR
EXTERN EndDialog@8:NEAR
EXTERN SendDlgItemMessageA@20:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;файл process.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include process.inc
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\psapi.lib
;--------------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG     MSGSTRUCT <?>
      HINST   DD 0           ; дескриптор приложения
      PA      DB "DIAL1",0
      PNAME   DB 300 DUP(0)
      BUF     DB 512 DUP(0)
      FORM    DB "%lu;%s",0
      HWN     DD ?
      PR_ID   DD 1000 DUP(0) ; массив идентификаторов процессов
      NB      DD 0
      HP      DD 0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
```

```
START:
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
;--------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  HINST
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
      JNE   KOL
;сообщение об ошибке
KOL:
;---------------------------------
      PUSH  0
      CALL  ExitProcess@4
;---------------------------------
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;----------------
      CMP  DWORD PTR [EBP+0CH],WM_COMMAND
      JNE  L3
;кнопка?
      CMP  WORD PTR [EBP+10H],102
      JE   L2
L3:
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
```

```
      CALL EndDialog@8
      JMP  FINISH
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  FINISH
;запомним дескриптор окна
      MOV  EAX,DWORD PTR [EBP+08H]
      MOV  HWN,EAX
L2:
;очистить список
      PUSH 0
      PUSH 0
      PUSH LB_RESETCONTENT
      PUSH 101
      PUSH HWN
      CALL SendDlgItemMessageA@20
;вызвать функцию EnumProcesses
      PUSH OFFSET NB
      PUSH 4000
      PUSH OFFSET PR_ID
      CALL EnumProcesses@12
;вывод процессов
      CALL ADD_PROCESS
FINISH:
      MOV EAX,0
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      RET 16
WNDPROC  ENDP
;процедура вывода в ListBox
ADD_PROCESS PROC
      XOR  EDI,EDI
LL:
      CMP  NB,0
      JZ   LL1
;обнулить буфер для имени процесса (имени приложения)
      MOV  PNAME,0
;открыть процесс
      PUSH DWORD PTR [PR_ID+EDI]
      PUSH 0
      PUSH PROCESS_QUERY_INFORMATION or PROCESS_VM_READ
```

```
      CALL OpenProcess@12
      MOV  HP,EAX
      CMP  EAX,0
      JZ   LL2
;процесс открыт, попробуем получить имя файла
      PUSH 300
      PUSH OFFSET PNAME
      PUSH 0
      PUSH HP
      CALL GetModuleFileNameExA@16
LL2:
;сформировать строку для списка
      PUSH OFFSET PNAME
      PUSH DWORD PTR [PR_ID+EDI]
      PUSH OFFSET FORM
      PUSH OFFSET BUF
      CALL wsprintfA
;освобождаем стек
      ADD  ESP,16
;добавить в список
      PUSH OFFSET BUF
      PUSH 0
      PUSH LB_ADDSTRING
      PUSH 101
      PUSH HWN
      CALL SendDlgItemMessageA@20
;закрыть процесс
      PUSH HP
      CALL CloseHandle@4
;к следующему процессу
      SUB  NB,4
      ADD  EDI,4
      JMP  LL
LL1:
      RET
ADD_PROCESS ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.5.8:

```
ml /c /coff  process.asm
rc process.rc
link /subsystem:windows process.obj process.res
```

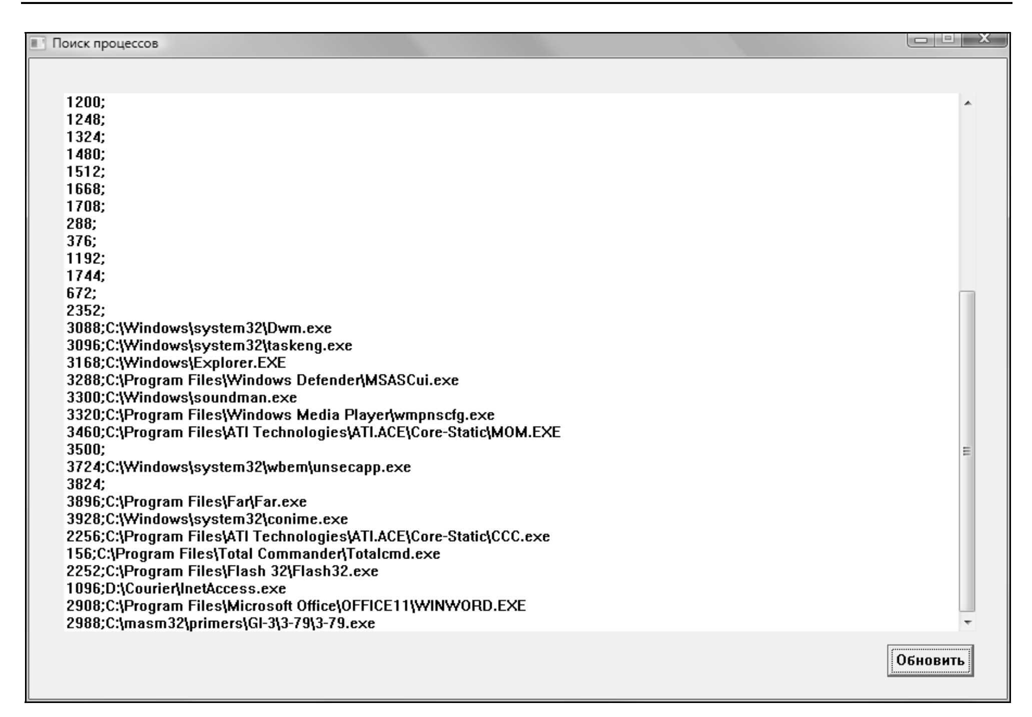
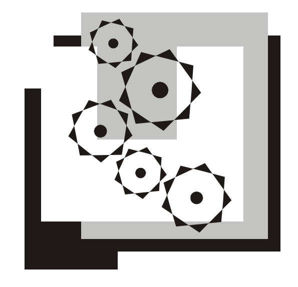


Рис. 3.5.4. Окно со списком процессов в системе

ЗАМЕЧАНИЕ

Получив с помощью функции `OpenProcess` дескриптор процесса, можно посредством функции `EnumProcessModules` получить список дескрипторов модулей данного процесса. Ну, а затем воспользоваться уже известной нам функцией `GetModuleFileNameEx`, которая возвращает полное имя модуля по его дескриптору. Мы использовали эту функцию, подставив вместо дескриптора модуля 0, и получили полное имя главного модуля процесса.



Глава 3.6

Некоторые вопросы системного программирования в Windows

Большая часть главы будет посвящена структуре и управлению памятью операционной системы Windows. Данный материал требует от читателя некоторой подготовки в области защищенного режима микропроцессоров Intel, и я излагаю основы этого режима. Более подробно о защищенном режиме можно узнать в книгах [1, 5] *(см. также приложение 3)*. Материалы данной главы пригодятся нам в дальнейшем, когда в *главе 4.6* мы будем рассматривать драйверы, работающие в режиме ядра.

Страничная и сегментная адресация

Начну изложение материала с некоторого исторического экскурса. Семейство микропроцессоров Intel[1] ведет свое начало с микропроцессора Intel 8086. В настоящее время во всю работает уже седьмое поколение. Каждое новое поколение отличалось от предыдущего в программном отношении, главным образом, расширением набора команд. Но были в этой восходящей лестнице и две ступени, сыгравшие огромную роль в развитии компьютеров на базе микропроцессоров Intel. Это микропроцессор 80286 (защищенный режим) и микропроцессор 80386 (переход на 32-битные регистры и страничная адресация).

До появления микропроцессора 80286 микропроцессоры использовались в так называемом *реальном режиме адресации*. Кратко изложу, в чем заключался этот режим. Для программирования использовался логический адрес, состоящий из двух 16-битных компонентов: сегмента и смещения. Сегментный адрес мог храниться в одном из четырех сегментных регистров: `CS`, `DS`,

---

[1] Я имею в виду и микропроцессоры, совместимые с Intel и выпускаемые другими фирмами.

SS, ES. Смещение хранилось в одном из индексных регистров: DI, SI, BX, BP, SP[2]. При обращении к памяти логический адрес подвергался преобразованию, заключающемуся в том, что к смещению прибавлялся сегментный адрес, сдвинутый на четыре бита влево. В результате получался 20-битный адрес, который, как легко заметить, мог охватывать всего около 1 Мбайт памяти[3] (точнее, 1087 Кбайт). Операционная система MS-DOS и была изначально рассчитана для работы в таком адресном пространстве. Получаемый 20-битный адрес назывался линейным, и при этом фактически совпадал с физическим адресом ячейки памяти. На рис. 3.6.1 схематически показан механизм преобразования логического в физический адрес реального режима микропроцессора Intel.

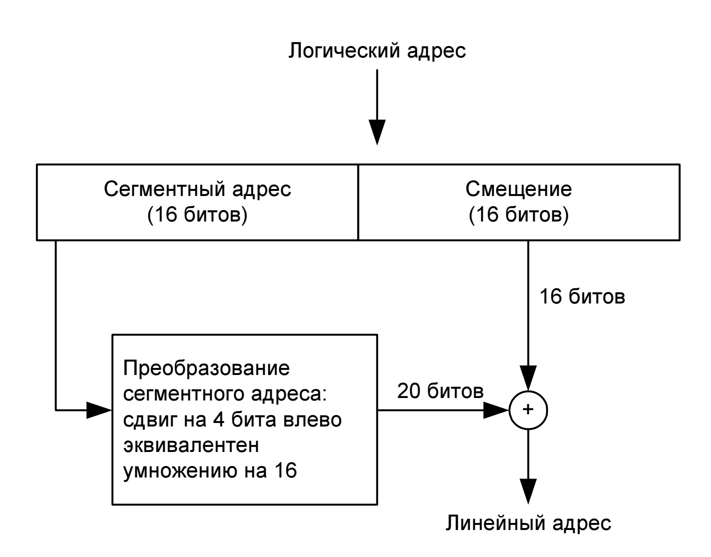


Рис. 3.6.1. Схема преобразования логического адреса в линейный адрес в реальном режиме адресации

Разумеется, с точки зрения развития операционных систем это был тупик. Должна быть, по крайней мере, возможность расширять память, и не просто расширять, а сделать все адресное пространство равноправным. Кроме этого, в реальном режиме вся память была доступна любому исполняемому приложению. Это касалось и памяти, используемой операционной системой. Любая ошибка (или злой умысел) программиста могла привести к остановке всей системы. Выход был найден с введением так называемого *защищенного режима*.

[2] В узком смысле слова индексными регистрами называются DI и SI.

[3] Когда-то казалось, что один мегабайт памяти — это много.

Гениальность подхода заключалась в том, что, на первый взгляд, ничего не изменилось. По-прежнему логический адрес формировался при помощи сегментных регистров и регистров, где хранилось смещение. Однако сегментные регистры хранили теперь не сегментный адрес, а так называемый селектор, часть которого (13 битов) представляла собой индекс в некоторой таблице, называемой дескрипторной. Индекс указывал на дескриптор, в котором хранилась полная информация о сегменте. Размер дескриптора был достаточен для адресации уже гораздо большего объема памяти.

На рис. 3.6.2 схематически представлен алгоритм преобразования логического адреса в линейный адрес. Правда, за основу мы взяли уже 32-битный микропроцессор, а не 16-битный, как было в начале. Таблица дескрипторов или таблица базовых адресов могла быть двух типов: глобальная (GDT) и локальная (LDT). Тип таблицы определялся вторым битом содержимого сегментного регистра. На расположение глобальной таблицы и ее размер указывал регистр `GDTR`. Предполагалось, что содержимое этого регистра после его загрузки не должно меняться. В глобальной дескрипторной таблице должны храниться дескрипторы сегментов, занятых операционной системой. Адрес локальной таблицы дескрипторов хранился в регистре `LDTR`. Предполагалось, что локальных дескрипторных таблиц может быть несколько — одна для каждой запущенной задачи. Тем самым уже на уровне микропроцессора закладывалась поддержка многозадачности. Размер регистра `GDTR` составляет 48 битов: 32 бита — адрес глобальной таблицы, 16 битов — размер.

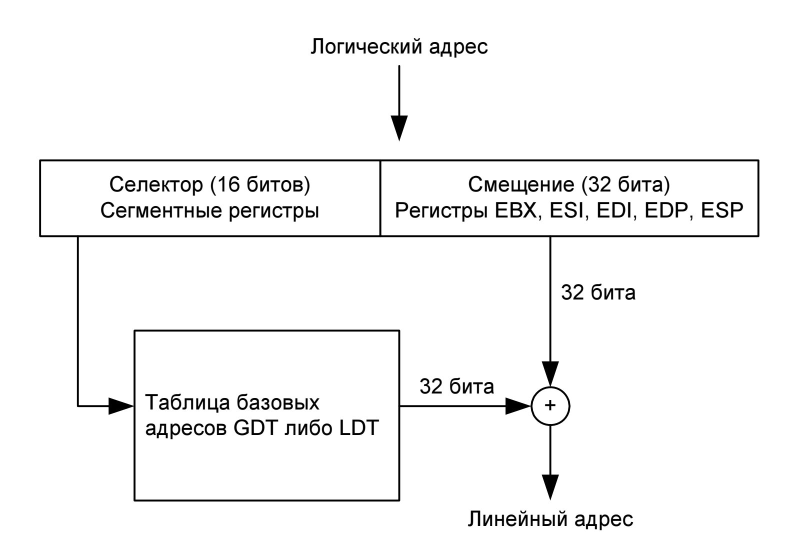


Рис. 3.6.2. Схема преобразования логического адреса в линейный адрес в защищенном режиме адресации

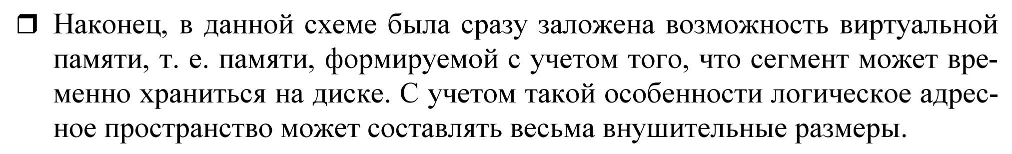
Кроме глобальной дескрипторной таблицы, предусматривалась еще одна общесистемная таблица — дескрипторная таблица прерываний (IDT). Она содержит дескрипторы специальных системных объектов, которые называются шлюзами и определяют точки входа процедур обработки прерываний и особых случаев. Положение дескрипторной таблицы прерываний определяется содержимым регистра `IDTR`, структура которого аналогична регистру `GDTR`.

Размер регистра `LDTR` составляет всего 10 байтов[4]. Первые 2 байта адресуют локальную дескрипторную таблицу не напрямую, а посредством глобальной дескрипторной таблицы, т. е. играют роль селектора для каждой вновь создаваемой задачи. Таким образом, в глобальную дескрипторную таблицу должен быть добавлен элемент, определяющий сегмент, где будет храниться локальная дескрипторная таблица данной задачи. Переключение же между задачами может происходить всего лишь сменой содержимого регистра `LDTR`. Отсюда, кстати, вытекает, что если задача одна собирается работать в защищенном режиме, то ей незачем использовать локальные дескрипторные таблицы и регистр `LDTR`.

Дескриптор сегмента содержал, в частности, поле доступа, которое определяло тип индексируемого сегмента (сегмент кода, сегмент данных, системный сегмент и т. д.). Здесь же можно, например, указать, что данный сегмент доступен только для чтения. Учитывалась также возможность, что сегмент может отсутствовать в памяти, т. е. временно находиться на диске. Тем самым закладывалась возможность реализовывать виртуальную память.

Подытожим, что же давал защищенный режим.

- Возможность для каждой задачи иметь свою систему сегментов. В микропроцессор закладывалась возможность быстрого переключения между задачами. Кроме того, предполагалось, что в системе будут существовать сегменты, принадлежащие операционной системе.
- Предполагалось, что сегменты могут быть защищены от записи.
- В поле доступа можно также указать уровень доступа. Всего возможно четыре уровня доступа. Смысл уровня доступа заключался в том, что задача не может получить доступ к сегменту, у которого уровень доступа выше, чем у данной задачи.



- Наконец, в данной схеме была сразу заложена возможность виртуальной памяти, т. е. памяти, формируемой с учетом того, что сегмент может временно храниться на диске. С учетом такой особенности логическое адресное пространство может составлять весьма внушительные размеры.

---

[4] В старых моделях микропроцессора Intel регистр содержал всего 2 байта.

Обратимся опять к рис. 3.6.2. Из схемы видно, что результатом преобразования является линейный адрес. Но если для микропроцессора 80286 линейный адрес можно отождествить с физическим адресом, для микропроцессора 80386 это уже не так.

Начиная с микропроцессора Intel 80386, появился еще один механизм преобразования адресов — это *страничная адресация*. Чтобы механизм страничной адресации заработал, старший бит системного регистра `CR0` должен быть равен 1.

Обратимся к рис. 3.6.3. Линейный адрес, получаемый путем дескрипторного преобразования (см. рис. 3.6.2), делится на три части. Старшие 10 битов адреса используются как индекс в таблице, которая называется каталогом таблиц страниц. Расположение каталога страниц определяется содержимым регистра `CR3`. Каталог состоит из дескрипторов. Максимальное количество дескрипторов — 1024. Самих же каталогов может быть бесчисленное множество, но в данный момент работает каталог, на который указывает регистр `CR3`.

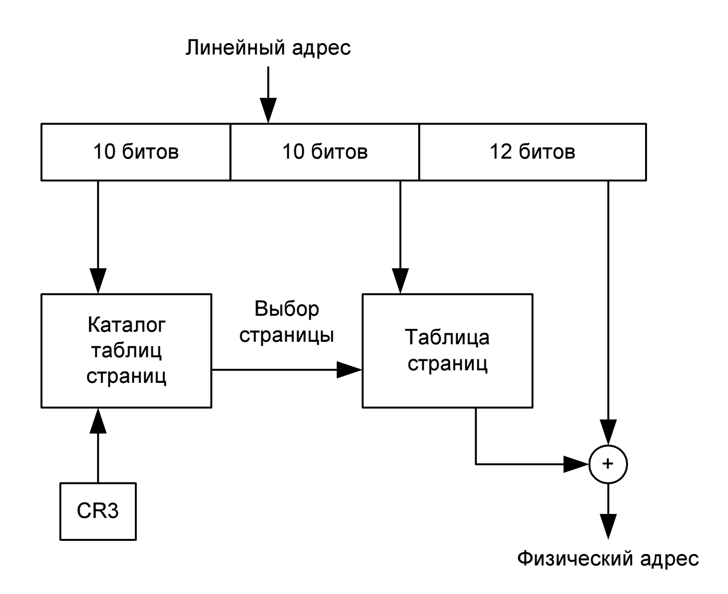


Рис. 3.6.3. Преобразование линейного адреса в физический адрес с учетом страничной адресации

Средние 10 битов линейного адреса предназначены для индексации таблицы страниц, которая содержит 1024 дескриптора страниц, а те, в свою очередь, определяют физический адрес страниц. Размер страницы обычно составляет 4 Кбайт. Легко сосчитать, какое адресное пространство может быть охвачено

одним каталогом таблиц страниц. Это составляет 1024×1024×1024×4 байтов, т. е. порядка 4 Гбайт.

Младшие 12 битов определяют смещение внутри страницы. Как легко заметить, это как раз составляет 4 Кбайт (4095 байтов). Конечно, читатель уже догадался, что для каждого процесса должен существовать свой каталог таблиц страниц. Переключение же между процессами можно осуществлять посредством изменения содержимого регистра `CR3`. Однако это не совсем рационально, т. к. требует большого объема памяти. В реальной ситуации для переключения между процессами производится изменение каталога таблиц страниц.

Обратимся теперь к структуре дескрипторов страниц (дескриптор таблицы страниц имеет ту же самую структуру):

- биты 12—31 — адрес страницы, который в дальнейшем складывается со смещением, предварительно сдвигаясь на двенадцать битов;
- биты 9—11 — для использования операционной системой;
- биты 7—8 — зарезервированы и должны быть равны нулю;
- бит 6 — устанавливается, если была осуществлена запись в каталог или страницу;
- бит 5 — устанавливается перед чтением и записью на страницу;
- бит 4 — запрет кэширования;
- бит 3 — бит сквозной записи;
- бит 2 — если значение этого бита равно 0, то страница относится к супервизору, если 1, то страница относится к рабочему процессу. Этим устанавливаются два уровня доступа;
- бит 1 — если бит установлен, то запись на страницу разрешена;
- бит 0 — если бит установлен, то страница присутствует в памяти. Страницы, содержащие данные, сбрасываются на диск и считываются, когда происходит обращение к ним. Страницы, содержащие код, на диск не сбрасываются, но могут подкачиваться из соответствующих модулей на диске. Поэтому память, занятая этими страницами, также может рационально использоваться.

Адресное пространство процесса

В предыдущем разделе мы говорили о страничной и сегментной адресации. Как же эти две адресации уживаются в Windows? Оказывается, все очень просто. В сегментные регистры загружаются селекторы, базовые адреса ко-

торых равны нулю, а размер сегмента составляет 4 Гбайт. После этого о существовании сегментов и селекторов можно забыть (в большей степени прикладному программисту), хотя для микропроцессора этот механизм попрежнему работает. Основным же механизмом формирования адреса становятся страничные преобразования. Такая модель памяти и называется *плоской* (flat). Логическая адресация в такой модели определяется всего одним 32-битным смещением. До сих пор все наши программы писались именно в плоской модели памяти. При этом мы представляли, что вся область памяти, адресуемая 32-битным адресом, находится в нашем распоряжении. Разумеется, мы были правы, только адрес этот является логическим адресом, который, в свою очередь, подвергается страничному преобразованию, а вот в какую физическую ячейку памяти он попадает, ответить уже весьма затруднительно.

На рис. 3.6.4 представлено логическое адресное пространство процесса. Особо обратите внимание на разделенные (совместно используемые) области памяти (области 2, 4, 5). Что это значит? А значит это только одно: эти области памяти проецируются на одно и то же физическое пространство. Рассмотрим назначение областей по порядку.

- Область 1. Эта область заблокирована и предназначена для выявления нулевых указателей. Особенно это относится к языку C, где функция `malloc` может возвращать нулевой указатель. Попытка записать по этому адресу приведет к сообщению операционной системы.
- Область 2. Эта область пространства использовалась в старых операционных системах Windows 9*x*. В операционных системах семейства Windows NT эта область входит в область 3. Для MS-DOS и 16-битных приложений здесь отводится свое адресное пространство.
- Следующая область адресного пространства (область 3), между 4 Мбайт и 2 Гбайт (в Windows 2000 и выше область начинается с 1 Мбайт), является адресным пространством процесса. Процесс занимает эту область пространства под код, данные, а также специфичные для него динамические библиотеки. Это неразделяемая область. Есть, однако, исключения, с которым мы уже встречались. Можно определить отдельные разделяемые секции. Это значит, что некоторые страницы из этого логического пространства будут отображаться в одну физическую область у разных процессов.
- Область 4. Закрытый раздел, используемый для внутренней реализации операционной системы.
- Следующая область (5) содержит в себе файлы, отображаемые в память, системные динамические библиотеки, а также динамическую память для

16-битных приложений. Для операционной системы Windows 2000 и выше эта область входит в следующую, шестую область.

- Последняя часть адресного пространства отведена под системные компоненты. Удивительно, но в Windows 9*x* эта область не была защищена от доступа обычных программ. В операционных системах семейства NT эта область недоступна исполняемым процессам.

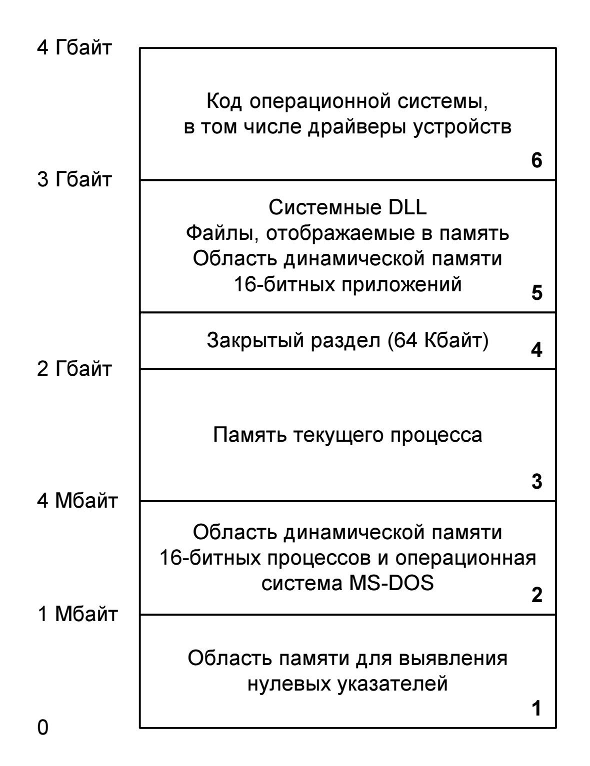


Рис. 3.6.4. Адресное пространство процесса

Для того чтобы в виртуальном адресном пространстве что-то хранилось, это пространство должно отображаться на физическую память. Физическая же страница памяти не всегда может находиться в оперативной памяти. Операционная система хранит часть страниц в страничном файле (pagefile.sys) или файлах, отображаемых в памяти *(см. главу 3.5)*. При обращении к адресу, относящемуся к странице, хранящейся на диске, возникает так называемое исключение, которое приводит к подкачке страницы с диска. В механизме участвуют так называемые свободные страницы. Система ищет свободную (не занятую никаким процессом) страницу и загружает туда необходимые

данные из страницы на диске, после чего страница оказывается занятой. Если свободных страниц нет, то система ищет страницу, которую можно выгрузить на диск, делает ее свободной и загружает данные в нее. Обычно, однако, операционная система заранее заботится, чтобы сводные страницы были в наличии.

Управление памятью

В *главе 3.5* мы уже рассматривали весьма эффективный способ использования памяти, имеющейся у процесса. Использование файлов, отображаемых в память, сводит обработку данных к обработке области памяти, включенной в единое с приложением адресное пространство. Но адресное пространство можно использовать и другими способами. Остановимся подробнее на двух подходах.

Динамическая память

В этом разделе мы разберем несколько функций, позволяющих динамически выделять и удалять блоки памяти.

Операционная система Windows поддерживает области памяти в виде куч (heaps). Процесс может содержать несколько куч, из которых программно можно получать определенные объемы памяти. Куча весьма удобна для работы с множеством небольших блоков памяти, например связанных списков или деревьев. Куча является объектом ядра, а, следовательно, управляется при помощи дескриптора.

Каждый процесс имеет кучу по умолчанию, выделяемую при создании процесса. Дескриптор этой кучи можно получить с помощью функции `GetProcessHeap`. Функция не имеет параметров.

Для управления кучами в Windows имеется несколько API-функций. Начнем с функции `GlobalAlloc`. Другая функция, `LocalAlloc`, фактически полностью эквивалентна первой и сохранена только для совместимости со старыми приложениями. Функция имеет два аргумента. Первым аргументом является флаг, о значении которого будем говорить далее. Вторым аргументом является число необходимых байтов выделяемой памяти. Если функция выполнена успешно, то она возвращает адрес начала блока, который можно использовать в дальнейших операциях. Если же система не может выделить достаточно памяти, то функция возвращает 0. В действительности функция `GlobalAlloc` выделяет память из собственной кучи процесса.

Обычно значение флага принимают равным константе `GMEM_FIXED`, которая равна нулю. Это означает, что блок памяти неперемещаемый. Неперемещае-

мость следует понимать в том смысле, что не будет меняться виртуальный адрес блока, тогда как адрес физической памяти, куда проецируется данный блок, может, разумеется, меняться системой. Комбинация данного флага с флагом `GMEM_ZEROINIT` приводит к автоматическому заполнению выделенного блока нулями, что часто бывает весьма удобно. Изменить размер выделенного блока можно при помощи функции `GlobalReAlloc`. Первым аргументом данной функции является указатель на изменяемый блок, второй аргумент — размер нового блока, третий аргумент — флаг. Замечу, что данная функция может изменить свойства блока памяти, т. е., например, сделать его перемещаемым.

Обратимся теперь снова к флагам функции `GlobalAlloc`. Дело в том, что если ваша программа интенсивно работает с памятью, т. е. многократно выделяет и освобождает память, память может оказаться фрагментированной. Действительно, вы же запрещаете перемещать блоки. В этом случае можно использовать флаг `GMEM_MOVEABLE`. Выделив блок, вы можете в любой момент зафиксировать его при помощи функции `GlobalLock`, после этого спокойно работая с ним. С помощью функции `GlobalUnlock` можно в любой момент снять фиксацию, т. е. разрешить системе упорядочивать блоки. Надо иметь в виду, что при использовании флага `GMEM_MOVEABLE` возвращается не адрес, а дескриптор. Но как раз аргументом функции `GlobalLock` и является дескриптор. Сама же функция `GlobalLock` возвращает адрес.

Возможен и еще "более экзотический" подход с использованием флага `GMEM_DISCARDABLE`. Этот флаг используется совместно с `GMEM_MOVEABLE`. В этом случае блок может быть удален из памяти системой, если только вы его предварительно не зафиксировали. Если блок был удален системой, то функция `GlobalLock` возвратит 0, и вам придется снова выделять блок и загружать данные, если необходимо.

Для удаления блока памяти используется функция `GlobalFree`. Причем в случае выделения фиксированного блока памяти аргументом функции является адрес блока памяти, а в случае перемещаемого блока памяти — дескриптор. Для освобождения удаляемого блока памяти используйте функцию `GlobalDiscard`.

В листинге 3.6.1 показано простейшее применение функции `GlobalAlloc`. Программа запрашивает блок памяти, считывает туда файл, а затем выводит содержимое этого файла в консольное окно. Имя файла следует указать в качестве параметра командной строки.

Листинг 3.6.1. Пример программы с распределением динамической памяти

```
;файл MEM.ASM
.586P
;плоская модель памяти
```

```
.MODEL FLAT, stdcall
;константы
;для вывода в консоль
STD_OUTPUT_HANDLE   equ -11
GENERIC_READ        equ 80000000h
OPEN_EXISTING       equ 3
;прототипы внешних процедур
EXTERN  GlobalFree@4:NEAR
EXTERN  GlobalAlloc@8:NEAR
EXTERN  GetFileSize@8:NEAR
EXTERN  CloseHandle@4:NEAR
EXTERN  CreateFileA@28:NEAR
EXTERN  ReadFile@20:NEAR
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetCommandLineA@0:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
      LENS  DWORD ?
      HANDL DWORD ?  ; дескриптор консоли
      HF    DWORD ?  ; дескриптор файла
      SIZEH DWORD ?  ; старшая часть длины файла
      SIZEL DWORD ?  ; младшая часть длины файла
      GH    DWORD ?  ; указатель на блок памяти
      NUMB  DWORD ?
      BUF   DB 10 DUP(0)
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить HANDLE вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;получить количество параметров
      CALL NUMPAR
      CMP  EAX,2
      JB  _EXIT
;--------------------------------------
;получить параметр номером EDI
      MOV  EDI,2
```

```
      LEA  EBX,BUF
      CALL GETPAR
;теперь работаем с файлом
;открыть только для чтения
      PUSH 0
      PUSH 0
      PUSH OPEN_EXISTING
      PUSH 0
      PUSH 0
      PUSH GENERIC_READ
      PUSH OFFSET BUF
      CALL CreateFileA@28
      CMP  EAX,-1
      JE   _EXIT
;запомнить дескриптор файла
      MOV  HF,EAX
;определить размер файла
      PUSH OFFSET SIZEH
      PUSH EAX
      CALL GetFileSize@8
;запомнить размер, предполагаем, что размер не превосходит 4 Гбайт
      MOV SIZEL,EAX
;запросить память для считывания туда файла
      PUSH EAX
      PUSH 0
      CALL GlobalAlloc@8
      CMP  EAX,0
      JE   _CLOSE
;запомнить адрес выделенного блока
      MOV  GH,EAX
;читать файл в выделенную память
      PUSH 0
      PUSH OFFSET NUMB
      PUSH SIZEL
      PUSH GH    ; адрес буфера
      PUSH HF
      CALL ReadFile@20
      CMP  EAX,0
      JE   _FREE
;вывести прочитанное
      PUSH 0
      PUSH OFFSET LENS
      PUSH SIZEL
      PUSH GH
      PUSH HANDL
      CALL WriteConsoleA@20
```

```
_FREE:
;освободить память
      PUSH GH
      CALL GlobalFree@4
;закрыть файлы
_CLOSE:
      PUSH HF
      CALL CloseHandle@4
_EXIT:
;конец работы программы
      PUSH 0
      CALL ExitProcess@4
;--------------------------------------------
;область процедур
;процедура определения количества параметров в строке
;определить количество параметров (->EAX)
NUMPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX ; указатель на строку
      XOR  ECX,ECX ; счетчик
      MOV  EDX,1   ; признак
L1:
      CMP  BYTE PTR [ESI],0
      JE   L4
      CMP  BYTE PTR [ESI],32
      JE   L3
      ADD  ECX,EDX  ; номер параметра
      MOV  EDX,0
      JMP  L2
L3:
      OR   EDX,1
L2:
      INC  ESI
      JMP  L1
L4:
      MOV  EAX,ECX
      RET
NUMPAR ENDP
;получить параметр из командной строки
;EBX указывает на буфер, куда будет помещен параметр
;в буфер помещается строка с нулем на конце
;EDI - номер параметра
GETPAR PROC
      CALL GetCommandLineA@0
      MOV  ESI,EAX   ; указатель на строку
      XOR  ECX,ECX   ; счетчик
```

```
      MOV  EDX,1     ; признак
L1:
      CMP  BYTE PTR [ESI],0
      JE   L4
      CMP  BYTE PTR [ESI],32
      JE   L3
      ADD  ECX,EDX   ; номер параметра
      MOV  EDX,0
      JMP  L2
L3:
      OR   EDX,1
L2:
      CMP  ECX,EDI
      JNE  L5
      MOV  AL,BYTE PTR [ESI]
      MOV  BYTE PTR [EBX],AL
      INC  EBX
L5:
      INC  ESI
      JMP  L1
L4:
      MOV  BYTE PTR [EBX],0
      RET
GETPAR ENDP
_TEXT ENDS
END START
```

Трансляция программы, представленной в листинге 3.6.1:

```
ML /c /coff MEM.ASM
LINK /SUBSYSTEM:CONSOLE MEM.OBJ
```

В листинге 3.6.1 описаны и использованы API-функции, имена которых начинаются с префикса `Global`. В документации Microsoft рекомендуется в новых приложениях отказываться от этих функций и использовать функции, начинающиеся с префикса `Heap`. Кратко опишем некоторые из этих функций.

- Функция `HeapCreate`. Создает новую кучу. Параметры функции:
  - 1-й параметр содержит флаги, описывающие свойства создаваемой кучи. Положив значение флага равным 0, мы тем самым указываем, что доступ к куче разных потоков будет последовательным;
  - 2-й параметр определяет количество байтов, первоначально передаваемых куче. Система округляет это значение до значения, кратного размеру страницы;
  - 3-й параметр указывает максимальное значение, до которого может увеличиваться куча.

- Функция `HeapAlloc`. Выделение блока памяти из кучи. Предпочтительнее использовать эту функцию вместо функции `GlobalAlloc`. Функция возвращает адрес выделенного блока. Параметры функции:
  - 1-й параметр — дескриптор кучи;
  - 2-й параметр содержит флаги, влияющие на характер выделения памяти. Например, использование флага `HEAP_ZERO_MEMORY=8h` приводит к тому, что в выделяемом блоке памяти обнуляются все байты;
  - 3-й параметр содержит количество выделяемых в куче байтов.
- Функция `HeapReAlloc`. Изменяет размер выделенного блока. Параметры функции:
  - 1-й параметр — дескриптор кучи;
  - 2-й параметр — флаги. Можно использовать те же флаги, что при выделении блока;
  - 3-й параметр — адрес блока;
  - 4-й параметр — количество байтов в измененном блоке.
- Функция `HeapDestroy`. Уничтожает кучу. Единственным параметром функции является дескриптор кучи.
- Функция `HeapFree`. Функция освобождает блок. Параметры функции:
  - 1-й параметр — дескриптор кучи;
  - 2-й параметр — обычно 0;
  - 3-й параметр — адрес блока.

В листинге 3.6.2 представлен фрагмент программы, который выполняет такие же действия, что и аналогичный фрагмент из листинга 3.6.1. Но здесь используются функции с префиксом `Heap`. Кроме этого, обратите внимание, что для того чтобы использовать эти функции, мы предварительно с помощью функции `GetProcessHeap` получили дескриптор собственной кучи процесса.

Листинг 3.6.2. Фрагмент кода с использованием функций с префиксом `Heap`

```
;открыть только для чтения
      PUSH 0
      PUSH 0
      PUSH OPEN_EXISTING
      PUSH 0
      PUSH 0
      PUSH GENERIC_READ
      PUSH OFFSET BUF
      CALL CreateFileA@28
```

```
      CMP  EAX,-1
      JE  _EXIT
;запомнить дескриптор файла
      MOV  HF,EAX
;определить размер файла
      PUSH OFFSET SIZEH
      PUSH EAX
      CALL GetFileSize@8
;запомнить размер, предполагаем, что размер не превосходит 4 Гбайт
      MOV   SIZEL,EAX
;получить дескриптор собственной кучи
      CALL  GetProcessHeap@0
      MOV   HANDL1,EAX
;запросить память для считывания туда файла
      PUSH  SIZEL
      PUSH  8 ;обнулить байты
      PUSH  HANDL1
      CALL  HeapAlloc@12
      CMP   EAX,0
      JE    _CLOSE
;запомнить адрес выделенного блока
      MOV   GH,EAX
;читать файл в выделенную память
      PUSH  0
      PUSH  OFFSET NUMB
      PUSH  SIZEL
      PUSH  GH ;адрес буфера
      PUSH  HF
      CALL  ReadFile@20
      CMP   EAX,0
      JE    _FREE
;вывести прочитанное
      PUSH  0
      PUSH  OFFSET LENS
      PUSH  SIZEL
      PUSH  GH
      PUSH  HANDL
      CALL  WriteConsoleA@20
_FREE:
;освободить память
      PUSH GH
      PUSH 0
      PUSH HANDL1
      CALL HeapFree@12
;закрыть файлы
_CLOSE:
      PUSH HF
      CALL CloseHandle@4
```

Виртуальная память

Операционная система Windows предоставляет также группу функций, осуществляющих управление виртуальной памятью. Виртуальная память удобна для работы с большими массивами данных.

Основной функцией для управления виртуальной памятью является функция `VirtualAlloc`. Вот параметры этой функции:

- 1-й параметр — адрес блока памяти для резервирования или передачи ему физической памяти. Обычно полагают его равным 0, т. е. предлагая системе самой определиться, в какой области адресного пространства выделить память;
- 2-й параметр — размер блока[5], который всегда должен быть кратен размеру страницы;
- 3-й параметр может быть равен:
  - `MEM_RESERVE = 2000h` — для резервирования блока;
  - `MEM_COMMIT = 1000h` — для резервирования и передачи ему физической памяти;
  - `MEM_PHYSICAL = 400000h` — выделить физическую память;
  - `MEM_RESET = 80000h` — сообщает системе, что данные, содержащиеся в блоке, более не понадобятся;
  - `MEM_TOP_DOWN = 100000h` — указать системе, что резервировать надо в старших адресах;
  - `MEM_WRITE_WATCH = 200000h` — система помечает страницы, куда производилась запись в данном блоке памяти;
- 4-й параметр определяет уровень защиты блока. Он может быть, например, равен `PAGE_READONLY = 2`, `PAGE_READWRITE = 4`, `PAGE_EXECUTE = 10h` или другой константе, определенной в документации Windows.

Возвращает функции виртуальный адрес блока памяти.

Суть данной функции заключается в том, что вы можете зарезервировать блок памяти, который не спроецирован на физическую память, а затем сделать так, чтобы этот блок (или часть его) был спроецирован на физическую память. После чего этот блок памяти можно уже использовать. Имейте в виду, что функция будет выделять блок памяти, кратный 64 Кбайт (точнее, размеру страницы), с адресом, также кратным 64 Кбайт.

---

[5] Строго говоря, речь должна идти не о блоках, а о регионах виртуальной памяти (range). Регионам же можно передавать блоки физической памяти. Мы, однако, не будем останавливаться на таких тонкостях.

Другая функция, `VirtualFree`, может освобождать блоки, задействованные функцией `VirtualAlloc`. Первым параметром этой функции является адрес блока. Вторым параметром функции является размер освобождаемого блока. Третий параметр функции может принимать значение `MEM_DECOMMIT` либо значение `MEM_RELEASE`. В первом случае блок (или его часть) перестает быть отображаемым. Во втором случае весь блок перестает быть зарезервированным. При этом значении второй параметр обязательно должен быть равен нулю.

Особо хочу отметить функцию `GlobalMemoryStatus`, с помощью которой можно определить количество свободной памяти. Единственным параметром данной функции является указатель на структуру, содержащую информацию о памяти. Вот эта структура:

```
MEM STRUC
      DwLength        DD ?
      DwMemoryLoad    DD ?
      DwTotalPhys     DD ?
      DwAvailPhys     DD ?
      DwTotalPageFile DD ?
      DwAvailPageFile DD ?
      DwTotalVirtual  DD ?
      DwAvailVirtual  DD ?
MEM ENDS
```

Здесь:

- `DwLength` — размер структуры в байтах;
- `DwMemoryLoad` — процент использованной памяти;
- `DwTotalPhys` — общий объем физической памяти в байтах;
- `DwAvailPhys` — объем доступной физической памяти в байтах;
- `DwTotalPageFile` — количество сохраненных байтов физической памяти на диске;
- `DwAvailPageFile` — количество доступных байтов памяти, сохраненных на диске;
- `DwTotalVirtual` — общий объем виртуальной памяти;
- `DwAvailVirtual` — объем доступной виртуальной памяти.

Фильтры (HOOKS)

Мы рассмотрим весьма эффективное средство, чаще всего используемое для отладки программ. Средство это называют *фильтрами* или *ловушками*[6]. Смысл его заключается в том, что вы при желании можете отслеживать

[6] Hook можно перевести как ловушка, да и по смыслу это ближе к понятию "ловушка".

сообщения как в рамках одного приложения, так и в рамках целой системы. В этой связи фильтры делят на глобальные (в рамках всей системы) и локальные (в рамках данного процесса). Работая с фильтрами, надо иметь в виду, что они могут существенным образом затормозить работу всей системы. Особенно это касается глобальных фильтров. С точки зрения программирования мы просто определяем функцию, которая вызывается системой при возникновении некоторого события. Можно также говорить о сообщении, приходящем в функцию фильтра.

Рассмотрим некоторые средства для работы с фильтрами. Далее перечислены основные типы фильтров или сообщения:

- `WH_CALLWNDPROC` — фильтр срабатывает, когда вызывается функция `SendMessage`;
- `WH_CALLWNDPROCRET` — фильтр срабатывает, когда функция `SendMessage` возвращает управление;
- `WH_CBT` — сообщение приходит, когда что-то происходит с окном;
- `WH_DEBUG` — данное сообщение посылается перед тем, как будет послано сообщение какому-либо другому фильтру;
- `WH_GETMESSAGE` — данный фильтр срабатывает, когда функция `GetMessage` принимает какое-либо сообщение из очереди;
- `WH_JOURNALRECORD` — данное сообщение приходит в процедуру фильтра, когда система удаляет из очереди какое-либо сообщение;
- `WH_JOURNALPLAYBACK` — вызывается после прихода сообщения `WH_JOURNALRECORD`;
- `WH_KEYBOARD` — сообщение приходит, когда происходят клавиатурные события;
- `WH_MOUSE` — аналогично предыдущему, но относится к событиям с мышью;
- `WH_MSGFILTER` — вызывается в случае событий ввода, которые произошли с диалоговым окном, меню, полосой прокрутки, но до того, как эти события были обработаны в пределах данного процесса;
- `WH_SHELL` — данный фильтр срабатывает, когда что-то происходит с Windows-оболочкой;
- `WH_SYSMSGFILTER` — аналогично сообщению `WH_MSGFILTER`, но относится ко всей системе;
- `WH_FOREGROUNDIDLE` — данный фильтр реагирует на событие, заключающееся в том, что наиболее приоритетный поток приложения переходит в состояние ожидания.

Фильтр устанавливается при помощи функции SetWindowsHookEx. Рассмотрим параметры этой функции:

- 1-й параметр — тип фильтра из тех, что перечислены выше;
- 2-й параметр — адрес процедуры фильтра. Если вы создаете для всей системы, то эта процедура должна находиться в динамической библиотеке. Исключение составляют лишь два типа фильтра: WH_JOURNALRECORD и WH_JOURNALPLAYBACK;
- 3-й параметр — дескриптор динамической библиотеки, если фильтр предназначен для всей системы. Исключение составляют два, уже упомянутых типа фильтра;
- 4-й параметр — идентификатор потока, если вы хотите следить за одним из потоков. Если значение этого параметра равно нулю, то создается фильтр для всей системы. Вообще говоря, поток может относиться и к вашему, и к "чужому" процессу.

Функция SetWindowsHookEx возвращает дескриптор фильтра.

Функция фильтра получает три параметра. Первый параметр определяет произошедшее событие в зависимости от типа фильтра. Два последующих параметра расшифровывают это событие. Поскольку для каждого типа фильтра может быть несколько событий, я не буду их перечислять. Их можно найти в справочном руководстве.

По окончании работы фильтр обязательно должен быть закрыт с помощью функции UnhookWindowsHookEx, единственным параметром которой является дескриптор фильтра.

Фильтр, вообще говоря, есть лишь некоторое звено в вызываемой системой цепочке, поэтому следует из своей процедуры фильтра вызвать функцию CallNextHookEx, которая передаст нужную информацию по цепочке. Таким образом, ответственность за то, чтобы вся цепочка функционировала верно, лежит полностью на разработчике.

Параметры функции CallNextHookEx:

- 1-й параметр — дескриптор вашего фильтра;
- 2-й, 3-й, 4-й параметры в точности соответствуют трем параметрам, переданным вашей процедуре фильтра.

В листинге 3.6.3 приводится пример простого фильтра, который отлавливает все произошедшие в системе нажатия клавиши <Пробел>. Обратите внимание, что поскольку устанавливаемый нами фильтр является глобальным, мы помещаем процедуру фильтра в динамическую библиотеку.

Листинг 3.6.3. Простой пример построения глобального фильтра

```
//файл dial.rc для программы DLLE.ASM
//определение констант
#define WS_SYSMENU       0x00080000L
#define WS_MINIMIZEBOX   0x00020000L
#define WS_MAXIMIZEBOX   0x00010000L
//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 120
STYLE WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
CAPTION "Пример программы с фильтром"
FONT 8, "Arial"
{
}
;основной модуль DLLE.ASM,
;устанавливающий фильтр в динамической библиотеке
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
;сообщение приходит при закрытии окна
WM_CLOSE       equ 10h
WM_INITDIALOG  equ 110h
WH_KEYBOARD    equ 2
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;прототипы внешних процедур
EXTERN  UnhookWindowsHookEx@4:NEAR
EXTERN  SetWindowsHookExA@16:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  GetProcAddress@8:NEAR
EXTERN  LoadLibraryA@4:NEAR
EXTERN  FreeLibrary@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  MessageBoxA@16:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
```

```
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG    MSGSTRUCT <?>
      HINST  DD 0 ;дескриптор приложения
      PA     DB "DIAL1",0
      LIBR   DB 'DLL2.DLL',0
      HLIB   DD ?
      APROC  DD ?
      HH     DD ?
      ATOH   DD ?
      NAMEPROC  DB '_HOOK@0',0
      NAMEPROC1 DB '_TOH@0',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;загрузить библиотеку
      PUSH OFFSET LIBR
      CALL LoadLibraryA@4
      CMP  EAX,0
      JE   _EXIT
      MOV  HLIB,EAX
;получить адрес процедуры-фильтра
      PUSH OFFSET NAMEPROC
      PUSH HLIB
      CALL GetProcAddress@8
      CMP  EAX,0
      JE   _EXIT
      MOV  APROC,EAX
;получить адрес вспомогательной процедуры
      PUSH OFFSET NAMEPROC1
      PUSH HLIB
      CALL GetProcAddress@8
      CMP  EAX,0
      JE   _EXIT
      MOV  ATOH,EAX
;здесь установить HOOK
      PUSH 0
      PUSH HLIB
      PUSH APROC
      PUSH WH_KEYBOARD
      CALL SetWindowsHookExA@16
      MOV  HH,EAX
;запомним и передадим в библиотеку
      MOV  EAX,ATOH
```

```
      PUSH HH
      CALL ATOH
;открыть диалоговое окно
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  [HINST]
CALL  DialogBoxParamA@20
;удалить HOOK
      PUSH HH
      CALL UnhookWindowsHookEx@4
;закрыть библиотеку
;библиотека автоматически закрывается также
;при выходе из программы
      PUSH OFFSET NAMEPROC
      PUSH HLIB
      CALL FreeLibrary@4
;выход
_EXIT:
      PUSH 0
      CALL ExitProcess@4
;процедура окна
;расположение параметров в стеке
; [BP+014H]  ;LPARAM
; [BP+10H]   ;WAPARAM
; [BP+0CH]   ;MES
; [BP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH EBX
      PUSH ESI
      PUSH EDI
;----------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP  FINISH
L1:
      CMP DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE FINISH
FINISH:
      POP EDI
```

```
      POP ESI
      POP EBX
      POP EBP
      MOV EAX,0
      RET 16
WNDPROC  ENDP
_TEXT ENDS
END START
;динамическая библиотека DLL2.ASM, содержащая процедуру-фильтр
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
PUBLIC HOOK, TOH
;константы
;сообщения, приходящие при открытии динамической библиотеки
DLL_PROCESS_DETACH  equ 0
DLL_PROCESS_ATTACH  equ 1
DLL_THREAD_ATTACH   equ 2
DLL_THREAD_DETACH   equ 3
;прототипы внешних процедур
EXTERN  CallNextHookEx@16:NEAR
EXTERN  MessageBoxA@16:NEAR
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
      HDL   DD ?
      HHOOK DD ?
      CAP   DB "Сообщение фильтра",0
      MES   DB "Нажат пробел",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
; [EBP+10H]   ;резервный параметр
; [EBP+0CH]   ;причина вызова
; [EBP+8]     ;идентификатор DLL-модуля
DLLENTRY:
      MOV EAX,DWORD PTR [EBP+0CH]
      CMP EAX,0
      JNE D1
;закрытие библиотеки
      JMP _EXIT
D1:
      CMP EAX,1
      JNE _EXIT
```

```
;открытие библиотеки
;запомнить идентификатор динамической библиотеки
      MOV EDX,DWORD PTR [EBP+08H]
      MOV HDL,EDX
_EXIT:
      MOV EAX,1
      RET 12
;--------------------
TOH PROC EXPORT
      PUSH  EBP
      MOV   EBP,ESP
      MOV   EAX,DWORD PTR [EBP+08H]
      MOV   HHOOK,EAX
      POP   EBP
      RET
TOH ENDP
;процедура фильтра
HOOK PROC EXPORT
      PUSH  EBP
      MOV   EBP,ESP
;отправить сообщение по цепочке
      PUSH DWORD PTR [EBP+010H]
      PUSH DWORD PTR [EBP+0CH]
      PUSH DWORD PTR [EBP+08H]
      PUSH HHOOK
      CALL CallNextHookEx@16
;проверить, не нажата ли клавиша <Пробел>
      CMP DWORD PTR [EBP+0CH],32
      JNE _EX
;нажата - выводим сообщение
      PUSH 0         ;MB_OK
      PUSH OFFSET CAP
      PUSH OFFSET MES
      PUSH 0         ;в окне экрана
      CALL MessageBoxA@16
_EX:
      POP  EBP
      RET
HOOK ENDP
_TEXT ENDS
END DLLENTRY
```

Трансляция программ на листинге 3.6.3:

- динамическая библиотека

```
ml /c /coff  dll2.asm
link /subsystem:windows /DLL dll2.obj
```

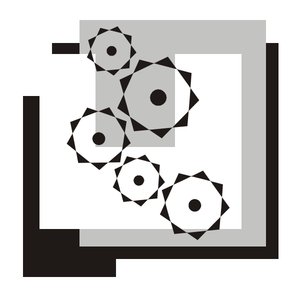
- основная программа

```
ml /c /coff  dllex.asm
rc dial.rc
link /subsystem:windows dllex.obj dial.res
```

При разборе программ в листинге 3.6.3 обратите внимание на роль, которую играет процедура `TOH`. Заметьте также, что второй и третий параметры процедуры фильтра в точности соответствует значениям аналогичных параметров сообщения `WM_KEYDOWN`. Кстати, надеюсь, вы понимаете, почему при нажатии клавиши <Пробел> появляются два сообщения — по одному на нажатие и отпускание.

Замечание

Фильтры дают нам еще один инструмент, позволяющий контролировать ввод в каком-либо из окон. Однако, согласитесь, что использовать данный инструмент для такого контроля слишком расточительно.



Глава 3.7

Совместное использование ассемблера с языками высокого уровня

Данная глава посвящена вопросам интеграции ассемблера с языками высокого уровня. Совместное использование ассемблера с языками высокого уровня может идти по трем направлениям:

- на основе объединения объектных модулей (раннее связывание);
- на основе динамических библиотек (позднее связывание);
- на основе встроенного языка ассемблера.

Обо всех трех направлениях пойдет речь в данной главе.

К сожалению, многие современные программисты, не зная языка ассемблера или не зная, как его использовать с языками высокого уровня, лишены мощного и гибкого инструмента программирования. Я бы сказал так: специалист по программированию на любом языке программирования должен владеть ассемблером как вторым инструментом. Это похоже на то, что изучение европейских языков в идеале должно предваряться изучением основ латыни. Радость общения с языком ассемблера велика, не упускайте этой возможности. Разве не для радости мы живем на этом свете?

Вообще, интеграция ассемблера с языками высокого уровня характеризуется тремя условиями: согласованием имен, согласованием параметров, согласованием вызовов. Остановимся сначала на последнем условии, т. е. на согласовании вызовов.

Согласование вызовов (исторический экскурс)

В древней операционной системе MS-DOS вызываемая процедура могла находиться либо в том же сегменте, что и команда вызова, тогда вызов назывался

близким (NEAR) или внутрисегментным, либо в другом сегменте, тогда вызов назывался дальним (FAR) или межсегментным. Разница заключалась в том, что адрес в первом случае формировался из двух байтов, а во втором — из четырех байтов. Соответственно, возврат из процедуры мог быть либо близким (RETN), т. е. адрес возврата формировался на основе двух байтов, взятых из стека, либо дальним (RETF), и в этом случае адрес формировался на основе четырех байтов, взятых опять же из стека. Ясно, что вызов и возврат должны быть согласованы друг с другом. В рамках единой программы это, как правило, не вызывало больших проблем. Но вот когда необходимо было подключить или какую-то библиотеку или объектный модуль, могли возникнуть трудности. Если в объектном модуле возврат осуществлялся по инструкции RETN, вы должны были компоновать объектные модули так, чтобы сегмент, где находится процедура, был объединен с сегментом, откуда осуществляется вызов. Вызов в этом случае, разумеется, должен быть близким (см. [1]). Если же возврат из процедуры осуществлялся по команде RETF, то и вызов этой процедуры должен быть дальним. При этом вызов и сама процедура при компоновке должны были попасть в разные сегменты. Проблема согласования дальности вызовов усугублялась еще и тем, что ошибки обнаруживались не при компоновке, а при исполнении программы. С этим были связаны и так называемые модели памяти в языке С, что также было головной болью многих начинающих программистов. Если, кстати, вы посмотрите на каталог библиотек С для DOS (наверное, кое-где они еще сохранились), то обнаружите, что для каждой модели памяти там существовала своя библиотека. Сегментация памяти приводила в С еще к одной проблеме — проблеме указателей, но это уже совсем другая история. В Турбо Паскале пошли по другому пути. Там приняли, что в программе должен существовать один сегмент данных и несколько сегментов кода. Если же вам не хватало одного сегмента для хранения данных, то предлагалось использовать динамическую память.

При переходе к операционной системе Windows мы получили замечательный подарок в виде плоской модели памяти. Теперь все вызовы являются близкими, т. е. осуществляющимися в пределах одного, но огромного сегмента. Тем самым снимается проблема согласования вызовов, и мы более к этой проблеме обращаться не будем.

Согласование имен

Согласование вызовов, как мы убедились, снято с повестки дня, а вот согласование имен год от года только усложнялось. Кое-что вы уже знаете. Транслятор MASM, как известно, если принята модель stdcall (Standard

Call, т. е. стандартный вызов), добавляет в конце имени `@N`, где `N` — количество передаваемых в стек параметров. То же по умолчанию делает и компилятор Visual C++ .

Другая проблема — символ подчеркивания перед именем. Транслятор MASM генерирует подчеркивание автоматически, если в начале программы устанавливается модель stdcall.

Еще одна проблема — согласование заглавных и прописных букв. Транслятор MASM делает это автоматически. Как известно, и в стандарте языка C с самого начала предполагалось различие между заглавными и прописными буквами. В Паскале же прописные и заглавные буквы не различаются. В этом есть своя логика: Турбо Паскаль и Delphi не создают стандартных объектных модулей, зато могут подключать их. При создании же динамических библиотек туда помещается имя так, как оно указано в заголовке процедуры.

Наконец, последняя проблема, связанная с согласованием имен, — это уточняющие имена в C++. Дело в том, что в C++ возможна так называемая перегрузка функций. Это значит, что одно и то же имя может относиться к разным функциям. В тексте программы эти функции отличаются по количеству и типу параметров и типу возвращаемого значения. Поэтому компилятор C++ автоматически делает в конце имени добавку — так, чтобы разные по смыслу функции различались при компоновке своими именами. Другими словами, имена в C++ искажаются. Разумеется, фирмы Borland и Microsoft и тут не пожелали согласовать свои позиции и делают в конце имени совершенно разные добавки. Обойти эту проблему не так сложно, нужно в программе на языке C для тех имен, к которым предполагается обращаться из других (внешних) модулей, использовать модификатор `EXTERN "C"`.

Согласование параметров

В табл. 3.7.1 представлены основные соглашения по передаче параметров в процедуру. Замечу в этой связи, что во всех наших ассемблерных программах мы указывали тип передачи параметров как stdcall. Однако, по сути, это никак и нигде не использовалось — так передача и извлечение параметров делалась нами явно, без помощи транслятора. Когда мы имеем дело с языками высокого уровня, это необходимо учитывать и знать, как работают те или иные соглашения.

Таблица 3.7.1 довольно ясно объясняет соглашения о передаче параметров (а также изменения в имени), и здесь более добавить нечего.

Таблица 3.7.1. Соглашения о вызовах

Соглашение	Параметры	Очистка стека	Регистры	Изменение имени
Pascal (конвенция языка Паскаль)	Слева направо	Процедура	Нет	
Register (быстрый — fastcall или регистровый вызов)	Слева направо	Процедура	Задействованы два регистра (ECX, EDX). Если для передачи параметров их не хватает, то остальные параметры передаются через стек	Префикс @. Суффикс @*N*
Cdecl (конвенция C)	Справа налево	Вызывающая программа	Нет	Префикс _
Stdcall (стандартный вызов)	Справа налево	Процедура	Нет	Префикс _. Суффикс @*N*

Остановлюсь еще на весьма важном моменте — типе возвращаемых функцией данных. С точки зрения ассемблера здесь все предельно просто: в регистре EAX возвращается значение, которое может быть либо числом, либо указателем на некую переменную или структуру. Если возвращаемое число имеет тип WORD, то оно передается в младшем слове регистра EAX. Наконец, если возвращается 64-битная величина, младшие 32 бита помещаются в регистр EAX, а старшие — в регистр EDX.

Однако, имея дело с C, вам надо очень аккуратно обращаться с такой проблемой, как преобразование типов. Преобразование типов — это целая наука, на которой мы не можем останавливаться в данной книге.

Простой пример использования ассемблера с языками высокого уровня

В данном разделе рассматривается простой модуль на языке ассемблера, содержащий процедуру, копирующую одну строку в другую. Мы подсоединяем этот модуль к программам, написанным на языках C и Паскаль, с использованием трансляторов: Visual C++ 2005 и Delphi 8.0.

Остановимся вначале на языке C++. Функцию, вызываемую из модуля, написанного на языке ассемблера, мы объявляем при помощи модификаторов

`extern "C"` и `C` (в ассемблерном модуле). Соглашение C предполагает, что стек освобождается в вызывающем модуле (см. табл. 3.7.1). В ассемблерном модуле вызываемая процедура должна быть дополнительно объявлена при помощи директивы `PUBLIC`. Модули на языке ассемблера и языке C++ представлены в листинге 3.7.1.

Листинг 3.7.1. Пример подключения объектного модуля к программе на языке C++ (тип согласования вызовов C)

```
//файл copyc.cpp
#include <stdio.h>
#include <windows.h>

extern "C"  char * COPYSTR(char *, char *);
void main()
{
char s1[100];
char *s2="Privet!";
printf("%s\n",COPYSTR(s1,s2));
printf("%s\n",s1);
ExitProcess(0);
}

;файл copy.asm
.586P
;плоская модель памяти
.MODEL FLAT, C
PUBLIC COPYSTR
_TEXT SEGMENT
;процедура копирования одной строки в другую
;строка, куда копировать [EBP+08H]
;строка, что копировать [EBP+0CH]
;не учитывает длину строки, куда производится копирование
COPYSTR PROC
      PUSH EBP
      MOV EBP,ESP
      MOV ESI,DWORD PTR [EBP+0CH]
      MOV EDI,DWORD PTR [EBP+08H]
L1:
      MOV AL,BYTE PTR [ESI]
      MOV BYTE PTR [EDI],AL
      CMP AL,0
      JE  L2
      INC ESI
      INC EDI
```

```
      JMP L1
L2:
      MOV EAX,DWORD PTR [EBP+08H]
      LEAVE
      RET
COPYSTR ENDP
_TEXT ENDS
END
```

Трансляция программы из листинга 3.7.1:

```
ml /c /coff  copy.asm
```

Далее транслируется программа на языке C++.

Комментарий к модулям в листинге 3.7.1.

- Для того чтобы скомпоновать объектный модуль в проект на языке Visual C++, следует обратиться к окну свойств проекта. На вкладке **Linker | Command Line** указать полное имя объектного модуля. В нашем случае я указал ..\..\..\copy.obj.
- Обратите внимание, как мы согласовали параметры. Выбрав тип согласования C, мы тем самым избавили себя от суффикса `@`, а во-вторых, нам не надо заботиться об очистке стека в ассемблерном модуле, потому что очистку осуществляет вызывающая сторона.

Зададимся теперь следующим вопросом. Как следует изменить ассемблерный модуль, если в нем установлено соглашение о вызовах stdcall? Порядок следования параметров не изменился, но MASM будет добавлять в конце имени суффикс `@`. В данном случае в модуле на языке C++ следует указать при объявлении функции `COPYSTR` тип вызова `__stdcall` (листинг 3.7.2).

Листинг 3.7.2. Пример подключения объектного модуля к программе на языке C++ (тип согласования вызовов stdcall)

```
//файл copyc.cpp
#include <stdio.h>
#include <windows.h>

extern "C"  char * __stdcall COPYSTR(char *, char *);
void main()
{
char s1[100];
char *s2="Privet!";
printf("%s\n",COPYSTR(s1,s2));
printf("%s\n",s1);
ExitProcess(0);
```

```
}

;файл copy.asm
.586P
.MODEL FLAT, stdcall
PUBLIC COPYSTR
;плоская модель памяти
_TEXT SEGMENT
;процедура копирования одной строки в другую
;строка, куда копировать [EBP+08H]
;строка, что копировать [EBP+0CH]
;не учитывает длину строки, куда производится копирование
;явное указывание параметров
COPYSTR PROC str1:DWORD, str2:DWORD
      MOV ESI,str2 ;DWORD PTR [EBP+0CH]
      MOV EDI,str1 ;DWORD PTR [EBP+08H]
L1:
      MOV AL,BYTE PTR [ESI]
      MOV BYTE PTR [EDI],AL
      CMP AL,0
      JE  L2
      INC ESI
      INC EDI
      JMP L1
L2:
      MOV EAX,DWORD PTR [EBP+08H]
RET
COPYSTR ENDP
_TEXT ENDS
END
```

Трансляция программы из листинга 3.7.2:

```
ml /c /coff  copy.asm
```

Далее транслируется программа на языке C++.

Комментарий к модулям из листинга 3.7.2. Для того чтобы в объектном модуле было указано правильное имя `_COPYSTR@8`, нам пришлось воспользоваться возможностями MASM и явно определить процедуру с указанием параметров. Теперь ассемблер сам организует, а затем освобождает стек процедуры.

Транслятор Delphi также вносит незначительные нюансы в данную проблему. Во-первых, для сегмента кода нам придется взять имя `CODE`. Во-вторых, из-за того, что Паскаль понимает строки несколько иначе, чем, скажем, C, в качестве строк мне пришлось взять символьный массив. Впрочем, опера-

тор `writeln` оказался довольно интеллектуальным и понял все с полуслова. Обратите внимание, что директива `stdcall` используется и в данном случае (листинг 3.7.3).

Листинг 3.7.3. Пример подключения объектного модуля к программе на Delphi

```
program Project2;
uses
  SysUtils;

{$APPTYPE CONSOLE}
{$L '..\copy.OBJ'}

function COPYSTR(s1,s2:PChar):PChar; stdcall; EXTERNAL ;
  var
  s1,s2:array[1..30] of char;
  s: PChar;
  begin
    s2[1]:='P';
    s2[2]:='r';
    s2[3]:='i';
    s2[4]:='v';
    s2[5]:='e';
    s2[6]:='t';
    s2[7]:=char(0);
    s:=COPYSTR(addr(s1[1]),addr(s2[1]));
    writeln(s);
    writeln(s1);
  end.

;файл copy.asm
.586P
.MODEL FLAT, Pascal
PUBLIC COPYSTR
;плоская модель памяти
CODE SEGMENT
;процедура копирования одной строки в другую
;строка, куда копировать [EBP+08H]
;строка, что копировать [EBP+0CH]
;не учитывает длину строки, куда производится копирование
COPYSTR PROC
      PUSH EBP
      MOV EBP,ESP
      MOV ESI,DWORD PTR [EBP+0CH]
      MOV EDI,DWORD PTR [EBP+08H]
```

```
L1:
      MOV AL,BYTE PTR [ESI]
      MOV BYTE PTR [EDI],AL
      CMP AL,0
      JE L2
      INC ESI
      INC EDI
      JMP L1
L2:
      MOV EAX,DWORD PTR [EBP+08H]
      POP EBP
      RET 8
COPYSTR ENDP
CODE ENDS
END
```

Трансляция программы из листинга 3.7.3:

```
ml /c copy.asm
```

Комментарий к модулям в листинге 3.7.3.

- В первую очередь обратим внимание, что теперь модуль на языке ассемблера транслируется без ключа `/coff`. Дело в том, что транслятор Delphi не понимает новый COFF-формат объектных модулей.
- И еще один интересный момент. В модуле Delphi мы указываем соглашение о вызове `stdcall` (т. е. параметры следуют справа налево), а в модуле на языке ассемблера — `Pascal`. Здесь все просто: в противном случае нам не удастся достигнуть согласования имен. В модуле же на языке ассемблера мы все равно принимаем параметры по соглашению `stdcall` (т. к. сами организуем это процесс). Мы могли бы сменить соглашение в модуле Delphi на `Pascal`, но тогда нам пришлось бы изменить порядок следования параметров в модуле copy.asm. Ну и, естественно, при соглашениях вызовов `stdcall` и `Pascal` процедура сама должна очищать стек от передаваемых параметров (см. команду `RET 8` в конце ассемблерной процедуры).

Передача параметров через регистры

В этом разделе используется другой тип вызова — быстрый, или регистровый. В соответствии с табл. 3.7.1, этот тип вызова предполагает, что три первых параметра будут передаваться в регистрах (`ECX`, `EDX`), а остальные в стеке, справа налево. При этом если стек был задействован, освобождение его возлагается на вызываемую процедуру. Есть еще один нюанс. В случае быстрого вызова транслятор C добавляет к имени значок `@` спереди, что мы естественно учитываем в ассемблерном модуле. Наконец, замечу, что данное правило

действует только для транслятора Visual C++. Другие трансляторы пользуются совсем другими протоколами. Кроме этого, к имени функции, которая будет вызываться по данному соглашению, следует добавить префикс `@`. Соответственно модули на языке C и ассемблера представлены в листинге 3.7.4.

Листинг 3.7.4. Пример регистрового соглашения вызова процедуры

```
//файл ADD.cpp
#include <windows.h>
#include <stdio.h>
//объявляется внешняя функция сложения четырех целых чисел
extern "C" DWORD __fastcall ADDD(DWORD, DWORD, DWORD, DWORD);
void main()
{
DWORD a,b,c,d;
a=1; b=2; c=3; d=4;
printf("%lu\n",ADDD(a,b,c,d));
ExitProcess(0);
}
;файл add.asm
.586P
.MODEL FLAT, Pascal
PUBLIC @ADDD@16
;плоская модель памяти
_TEXT SEGMENT
;процедура возвращает сумму четырех параметров
;передача параметров регистровая
;первые два параметра в регистрах ECX, EDX (a - ECX, b - EDX)
;третий параметр (c) в стеке, т. е. [EPB+08H]
;четвертый параметр (d) в стеке, т. е. [EPB+0CH]
@ADDD@16 PROC
      PUSH EBP
      MOV EBP,ESP
      ADD ECX,EDX
      ADD ECX,DWORD PTR [EBP+08H]
      ADD ECX,DWORD PTR [EBP+0CH]
      MOV EAX,ECX
      POP EBP
      RET 8
@ADDD@16 ENDP
_TEXT ENDS
END
```

Трансляция модулей из листинга 3.7.4:

```
Ml /c /coff  add.asm
```

И далее подключаем объектный модуль в проект Visual C++ (см. комментарий к листингу 3.7.4).

Прокомментирую листинг 3.7.4. Обратите внимание, как нам удалось согласовать имя вызывающей процедуры в вызываемом модуле. Модификатор `__fastcall` в модуле на языке С формирует имя вызываемой функции по следующей схеме `@`*имя*`@N`. В данном случае имя это `ADDD`. Таким образом, из модуля на языке С будет вызываться функция с именем `@ADDD@16`. Вот это имя мы и должны обеспечить в объектном модуле. Делается это весьма просто. В модуле на языке ассемблера указывается соглашение `Pascal`, а далее имя процедуры задается в том виде, как того требует модуль С (см. листинг 3.7.4). Такие трюки — дело обычное, когда вы хотите согласовать два модуля, написанные на различных языках.

Замечание

Компиляторы Borland C++ и Borland Delphi (в Delphi это соглашение вместо fastcall называется register) придерживаются совсем другого соглашения при регистровом вызове. Первые три параметра помещаются, соответственно, в регистры `EAX`, `EDX`, `ECX`, и только начиная с четвертого параметра, в качестве канала передачи используется стек. К сожалению, и другие производители компиляторов не придерживаются какого-то одного протокола быстрого (регистрового) вызова.

Вызовы API и ресурсы в ассемблерных модулях

В данном разделе я показываю, что вызываемый ассемблерный модуль может содержать не только какие-то вспомогательные процедуры, но и вызывать функции API и пользоваться ресурсами. Причем ресурсы, разумеется, являются для всех модулей проекта общими. Можно иметь несколько файлов, задающих ресурсы, но главное, чтобы не совпадали имена и идентификаторы (листинг 3.7.5).

Программа из листинга 3.7.5 создает диалоговое окно, а в нем поле с точным временем и датой. Обратите внимание на роль двух таймеров в приложении: первый формирует строку "дата-время", а второй помещает эту строку в поле диалогового окна.

Листинг 3.7.5. Консольная программа на С++ вызывает процедуру, определенную в ассемблерном модуле, которая, в свою очередь, работает в GUI-режиме

```
//модуль на языке C++ (консольная программа)
#include <windows.h>
#include <stdio.h>
```

```
//объявляется внешняя функция
extern "C" void __stdcall DIAL1();
void main()
{
//вызов процедуры из ассемблерного модуля
DIAL1();
ExitProcess(0);
}

//файл dialforc.rc
//определение констант
#define WS_SYSMENU     0x00080000L
//элементы на окне должны быть изначально видимы
#define WS_VISIBLE     0x10000000L
//бордюр вокруг элемента
#define WS_BORDER      0x00800000L
//при помощи клавиши <Tab> можно по очереди активизировать элементы
#define WS_TABSTOP     0x00010000L
//текст в окне редактирования прижат к левому краю
#define ES_LEFT        0x0000L
//стиль всех элементов на окне
#define WS_CHILD       0x40000000L
//запрещается ввод с клавиатуры
#define ES_READONLY    0x0800L
#define DS_3DLOOK      0x0004L
//определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 100
STYLE WS_SYSMENU | DS_3DLOOK
CAPTION "Диалоговое окно с часами и датой"
FONT 8, "Arial"
{
 CONTROL "", 1, "edit", ES_LEFT | WS_CHILD
 | WS_VISIBLE | WS_BORDER
 | WS_TABSTOP | ES_READONLY, 100, 5, 130, 12
}

;файл dialforc.inc
;константы
;сообщение приходит при закрытии окна
WM_CLOSE        equ 10h
;сообщение приходит при создании окна
WM_INITDIALOG   equ 110h
;сообщение приходит при событии с элементом в окне
WM_COMMAND      equ 111h
;сообщение от таймера
WM_TIMER        equ 113h
```

```
;сообщение посылки текста элементу
WM_SETTEXT      equ 0Ch
;прототипы внешних процедур
EXTERN  SendDlgItemMessageA@20:NEAR
EXTERN  wsprintfA:NEAR
EXTERN  GetLocalTime@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  SetTimer@16:NEAR
EXTERN  KillTimer@8:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DD ?
      MSMESSAGE  DD ?
      MSWPARAM   DD ?
      MSLPARAM   DD ?
      MSTIME     DD ?
      MSPT       DD ?
MSGSTRUCT ENDS
;структура данных дата-время
DAT STRUC
      year    DW ?
      month   DW ?
      dayweek DW ?
      day     DW ?
      hour    DW ?
      min     DW ?
      sec     DW ?
      msec    DW ?
DAT ENDS

;файл dialforc.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include dialforc.inc
PUBLIC DIAL1
;---------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG      MSGSTRUCT <?>
      HINST    DD 0 ;дескриптор приложения
      PA       DB "DIAL1",0
```

```
      TIM      DB "Дата %u/%u/%u   Время %u:%u:%u",0
      STRCOPY  DB 50 DUP(?)
      DATA     DAT <0>
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
DIAL1 PROC
      PUSH  EBP
      MOV   EBP,ESP
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
;создать диалоговое окно
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  HINST
      CALL  DialogBoxParamA@20
      CMP   EAX,-1
;-------------------------------
      POP   EBP
      RET
DIAL1 ENDP
;-------------------------------
;процедура окна
;расположение параметров в стеке
; [BP+014H]  ;LPARAM
; [BP+10H]   ;WAPARAM
; [BP+0CH]   ;MES
; [BP+8]     ;HWND
WNDPROC  PROC
      PUSH  EBP
      MOV   EBP,ESP
      PUSH  EBX
      PUSH  ESI
      PUSH  EDI
;----------------
      CMP   DWORD PTR [EBP+0CH],WM_CLOSE
      JNE   L1
;здесь реакция на закрытие окна
;удалить таймер 1
      PUSH  1 ;идентификатор таймера
      PUSH  DWORD PTR [EBP+08H]
      CALL  KillTimer@8
```

```
;удалить таймер 2
      PUSH  2 ;идентификатор таймера
      PUSH  DWORD PTR [EBP+08H]
      CALL  KillTimer@8
;закрыть диалог
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      JMP  FINISH
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  L2
;здесь начальная инициализация
;установить таймер 1
      PUSH 0    ;параметр = NULL
      PUSH 1000 ;интервал 1 секунда
      PUSH 1    ;идентификатор таймера
      PUSH DWORD PTR [EBP+08H]
      CALL SetTimer@16
;установить таймер 2
      PUSH OFFSET TIMPROC ;параметр != NULL
      PUSH 500            ;интервал 0.5 секунд
      PUSH 2              ;идентификатор таймера
      PUSH DWORD PTR [EBP+08H]
      CALL SetTimer@16
      JMP  FINISH
L2:
      CMP DWORD PTR [EBP+0CH],WM_TIMER
      JNE FINISH
;отправить строку в окно
      PUSH OFFSET STRCOPY
      PUSH 0
      PUSH WM_SETTEXT
      PUSH 1 ;идентификатор элемента
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
FINISH:
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      MOV EAX,0
      RET 16
WNDPROC  ENDP
;-------------------------------------------
;процедура таймера
```

```
;расположение параметров в стеке
; [BP+014H]  ;LPARAM - промежуток запуска Windows
; [BP+10H]   ;WAPARAM - идентификатор таймера
; [BP+0CH]   ;WM_TIMER
; [BP+8]     ;HWND
TIMPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
;получить локальное время
      PUSH OFFSET DATA
      CALL GetLocalTime@4
;получить строку для вывода даты и времени
      MOVZX EAX,DATA.sec
      PUSH  EAX
      MOVZX EAX,DATA.min
      PUSH  EAX
      MOVZX EAX,DATA.hour
      PUSH  EAX
      MOVZX EAX,DATA.year
      PUSH  EAX
      MOVZX EAX,DATA.month
      PUSH  EAX
      MOVZX EAX,DATA.day
      PUSH  EAX
      PUSH  OFFSET TIM
      PUSH  OFFSET STRCOPY
      CALL  wsprintfA
;восстановить стек
      ADD  ESP,32
      POP  EBP
      RET  16
TIMPROC  ENDP
_TEXT ENDS
END
```

Трансляция модулей из листинга 3.7.5:

```
ml /c /coff  dialforc.asm
rc dialforc.rc
```

А далее объектные модули dialforc.obj и dialforc.res должны быть включены в проект Visual C++, как это мы уже делали неоднократно. После этого транслируется весь проект.

Особенностью данного примера является то, что в проект на языке высокого уровня включаются два объектных модуля obj и res. Как видим, здесь прослеживается полная совместимость. В действительности мы могли бы поступить

и по-другому. Подготовить файл ресурсов в интегрированной среде Visual C++ и там же его откомпилировать. В любом случае в модуле на языке ассемблера мы могли бы обращаться к ресурсам.

Развернутый пример использования языков ассемблера и С

Здесь рассматривается пример простейшего калькулятора. Для ассемблера бывает сложно найти библиотеки с определенными процедурами, писать же все самому — не хватит времени. Предлагаемая схема очень проста. По сути, программа на языке С (или другом языке высокого уровня) является неким каркасом. Она вызывает процедуру на языке ассемблера, которая и выполняет основные действия. Кроме того, в С-модуль можно поместить и другие процедуры, которые легче написать на С и которые будут вызываться из ассемблера. Здесь как раз и приводится такой пример. В С-модуль я поместил процедуры, которые преобразуют строки в вещественные числа и выполняют над ними действия и затем преобразуют результат опять в строку. Тексты программных модулей представлены в листинге 3.7.6, окно результирующего приложения см. на рис. 3.7.1.

Листинг 3.7.6. С-модуль для программы простейшего калькулятора, компонуемый с ассемблерным модулем

```
//calcc.cpp
#include <windows.h>
#include <stdio.h>

//главная ассемблерная процедура
extern "C" void __stdcall MAIN1();
//следующие функции будут вызываться из ассемблерного модуля
//сложить
extern "C" void __stdcall  sum(char *, char *, char *);
//вычесть
extern "C" void __stdcall  su(char *, char *, char *);
//умножить
extern "C" void __stdcall mu(char *, char *, char *);
//разделить
extern "C" void __stdcall dii(char *, char *, char *);

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  MAIN1();
```

```
  return 0;
}
extern "C" void __stdcall sum(char * s1, char * s2, char * s)
{
  double f1,f2,f;
  f1=atof(s1); f2=atof(s2);
  f=f1+f2;
  sprintf(s,"%f",f);
  strcat(s," +");
  return;
}
extern "C" void __stdcall  su(char * s1, char * s2, char * s)
{
  float f1,f2,f;
  f1=atof(s1); f2=atof(s2);
  f=f1-f2;
  sprintf(s,"%f",f);
  strcat(s," -");
  return;
}
extern "C" void __stdcall mu(char * s1, char * s2, char * s)
{
  float f1,f2,f;
  f1=atof(s1); f2=atof(s2);
  f=f1*f2;
  sprintf(s,"%f",f);
  strcat(s," *");
  return;
}
extern "C" void __stdcall dii(char * s1, char * s2, char * s)
{
  float f1,f2,f;
  f1=atof(s1); f2=atof(s2);
  if(f2!=0)
  {
    f=f1/f2;
    sprintf(s,"%f",f);
    strcat(s," /");
  } else strcpy(s,"Ошибка деления");
  return;
}

//calc.rc
//определение констант
//стили окна
#define WS_SYSMENU     0x00080000L
```

```
#define WS_MINIMIZEBOX 0x00020000L
#define DS_3DLOOK      0x0004L
#define ES_LEFT        0x0000L
#define WS_CHILD       0x40000000L
#define WS_VISIBLE     0x10000000L
#define WS_BORDER      0x00800000L
#define WS_TABSTOP     0x00010000L
#define SS_LEFT        0x00000000L
#define BS_PUSHBUTTON  0x00000000L
#define BS_CENTER      0x00000300L
#define DS_LOCALEDIT   0x20L
#define ES_READONLY    0x0800L
#define WS_OVERLAPPED  0x0
#define WS_CAPTION     0x0C

//идентификаторы кнопок
#define IDC_BUTTON1  101
#define IDC_BUTTON2  102
#define IDC_BUTTON3  103
#define IDC_BUTTON4  104
DIAL1 DIALOG 0, 0, 170, 110
STYLE WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX
| DS_3DLOOK
CAPTION "Пример простейшего калькулятора"
FONT 8, "Arial"
{
 CONTROL "", 1, "edit", ES_LEFT | WS_CHILD | WS_VISIBLE
   | WS_BORDER | WS_TABSTOP, 9, 8, 128, 12
 CONTROL "", 2, "edit", ES_LEFT | WS_CHILD | WS_VISIBLE
   | WS_BORDER | WS_TABSTOP, 9, 27, 128, 12
 CONTROL "", 3, "edit", ES_LEFT | WS_CHILD | ES_READONLY
   | WS_VISIBLE | WS_BORDER | WS_TABSTOP, 9, 76, 127, 12
 CONTROL "+", IDC_BUTTON1, "button", BS_PUSHBUTTON
   | BS_CENTER | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 11,
   48, 15, 14
 CONTROL "-", IDC_BUTTON2, "button", BS_PUSHBUTTON
   | BS_CENTER | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 34,
   48, 15, 14
 CONTROL "*", IDC_BUTTON3, "button", BS_PUSHBUTTON
   | BS_CENTER | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 56,
   48, 15, 14
 CONTROL "/", IDC_BUTTON4, "button", BS_PUSHBUTTON
   | BS_CENTER | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 80,
   48, 15, 14
}

;calc.inc
```

```
;константы
;сообщение приходит при закрытии окна
WM_CLOSE       equ 10h
WM_INITDIALOG  equ 110h
WM_COMMAND     equ 111h
WM_GETTEXT     equ 0Dh
WM_SETTEXT     equ 0Ch

;прототипы внешних процедур
EXTERN  ExitProcess@4:NEAR
EXTERN  GetModuleHandleA@4:NEAR
EXTERN  DialogBoxParamA@20:NEAR
EXTERN  EndDialog@8:NEAR
EXTERN  SendDlgItemMessageA@20:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
      MSHWND     DWORD ?
      MSMESSAGE  DWORD ?
      MSWPARAM   DWORD ?
      MSLPARAM   DWORD ?
      MSTIME     DWORD ?
      MSPT       DWORD ?
MSGSTRUCT ENDS

;модуль calc.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
include calc.inc
EXTERN  sum@12:NEAR
EXTERN  su@12:NEAR
EXTERN  mu@12:NEAR
EXTERN  dii@12:NEAR
PUBLIC  MAIN1
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;------------------------------------------------
;сегмент данных
_DATA SEGMENT
      MSG MSGSTRUCT <?>
      HINST DD 0    ;дескриптор приложения
      PA  DB "DIAL1",0
      S1  DB 50 DUP(0)
      S2  DB 50 DUP(0)
```

```
      S  DB 50 DUP(0)
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
;процедура, вызываемая из С-модуля
MAIN1 PROC
;получить дескриптор приложения
      PUSH  0
      CALL  GetModuleHandleA@4
      MOV   HINST, EAX
;--------------------------------
      PUSH  0
      PUSH  OFFSET WNDPROC
      PUSH  0
      PUSH  OFFSET PA
      PUSH  HINST
      CALL  DialogBoxParamA@20
;--------------------------------
      RET
MAIN1 ENDP
;процедура окна
;расположение параметров в стеке
; [EBP+014H]  ;LPARAM
; [EBP+10H]   ;WAPARAM
; [EBP+0CH]   ;MES
; [EBP+8]     ;HWND
WNDPROC  PROC
      PUSH EBP
      MOV  EBP,ESP
;--------------------------------
      CMP  DWORD PTR [EBP+0CH],WM_CLOSE
      JNE  L1
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL EndDialog@8
      MOV  EAX,1
      JMP  FINISH
L1:
      CMP  DWORD PTR [EBP+0CH],WM_INITDIALOG
      JNE  L2
;здесь заполнить окна редактирования, если надо
      JMP FINISH
L2:
      CMP DWORD PTR [EBP+0CH],WM_COMMAND
      JNE FINISH
      CMP WORD PTR [EBP+10H],101
```

```
      JNE NO_SUM
;первое слагаемое
      PUSH OFFSET S1
      PUSH 50
      PUSH WM_GETTEXT
      PUSH 1
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
;второе слагаемое
      PUSH OFFSET S2
      PUSH 50
      PUSH WM_GETTEXT
      PUSH 2
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
;сумма
      PUSH OFFSET S
      PUSH OFFSET S2
      PUSH OFFSET S1
      CALL sum@12
;вывести сумму
      PUSH OFFSET S
      PUSH 50
      PUSH WM_SETTEXT
      PUSH 3
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
      JMP  FINISH
NO_SUM:
      CMP  WORD PTR [EBP+10H],102
      JNE  NO_SUB
;уменьшаемое
      PUSH OFFSET S1
      PUSH 50
      PUSH WM_GETTEXT
      PUSH 1
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
;вычитаемое
      PUSH OFFSET S2
      PUSH 50
      PUSH WM_GETTEXT
      PUSH 2
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
;разность
      PUSH OFFSET S
```

```
      PUSH OFFSET S2
      PUSH OFFSET S1
      CALL su@12
;вычислить разность
      PUSH OFFSET S
      PUSH 50
      PUSH WM_SETTEXT
      PUSH 3
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
      JMP  FINISH
;
NO_SUB:
      CMP  WORD PTR [EBP+10H],103
      JNE  NO_MULT
;первый множитель
      PUSH OFFSET S1
      PUSH 50
      PUSH WM_GETTEXT
      PUSH 1
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
;второй множитель
      PUSH OFFSET S2
      PUSH 50
      PUSH WM_GETTEXT
      PUSH 2
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
;произведение
      PUSH OFFSET S
      PUSH OFFSET S2
      PUSH OFFSET S1
      CALL mu@12
;вывести произведение
      PUSH OFFSET S
      PUSH 50
      PUSH WM_SETTEXT
      PUSH 3
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
      JMP  FINISH
;
NO_MULT:
      CMP  WORD PTR [EBP+10H],104
      JNE  FINISH
```

```
;делимое
      PUSH OFFSET S1
      PUSH 50
      PUSH WM_GETTEXT
      PUSH 1
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
;делитель
      PUSH OFFSET S2
      PUSH 50
      PUSH WM_GETTEXT
      PUSH 2
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
;деление
      PUSH OFFSET S
      PUSH OFFSET S2
      PUSH OFFSET S1
      CALL dii@12
;вывести результат деления
      PUSH OFFSET S
      PUSH 50
      PUSH WM_SETTEXT
      PUSH 3
      PUSH DWORD PTR [EBP+08H]
      CALL SendDlgItemMessageA@20
      JNE  FINISH
FINISH:
      MOV  EAX,0
      POP  EBP
      RET  16
WNDPROC ENDP
_TEXT ENDS
END
```

Трансляция модулей из листинга 3.7.6:

```
ml /c /coff  calc.asm
rc calc.rc
```

И далее модули calc.obj и calc.res должны быть указаны в проекте Visual C++, который затем компилируется.

Особенностью данного примера является то, что здесь имеется двунаправленное связывание. С одной стороны, функция `MAIN1`, объявленная в программе на языке С, затем связывается с функцией из ассемблерного модуля. С другой стороны, в ассемблерном модуле объявлены процедуры `sum`, `su`, `mu`,

`dii`, которые являются внешними. Они должны быть связаны с соответствующими функциями в модуле на языке C. Обратите внимание, какие имена мы дали этим процедурам в ассемблерном модуле — это очень важно (см. листинг 3.7.6).

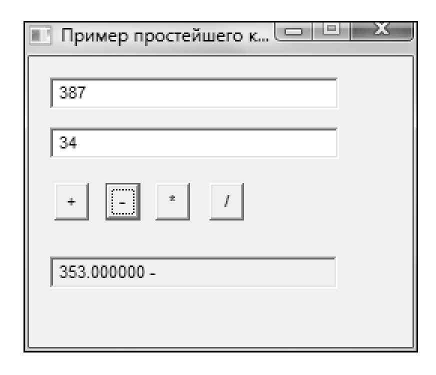


Рис. 3.7.1. Пример работы программы-калькулятора

Встроенный ассемблер

Теперь поговорим о встроенном ассемблере. Это весьма мощное средство. Надо только иметь в виду, что встроенные ассемблеры часто несколько отстают от обычных ассемблеров в поддержке новых команд микропроцессоров. Это вполне объяснимо, т. к. разработка новой версии пакета, скажем Visual C++, требует гораздо больше времени, чем пакета MASM32. В примерах из листингов 3.7.7 и 3.7.8 мы используем команды арифметического сопроцессора. Для выделения блока ассемблерных команд используются ключевые слова `asm` (Delphi) и `_asm` (Visual C++).

Листинг 3.7.7. Пример использования директивы `asm` и команд сопроцессора в программе на языке Паскаль (Delphi 8.0)

```
program Project1;
{$APPTYPE CONSOLE}
uses
SysUtils;
var
d:double;
{функция написана на встроенном ассемблере}
function soproc(f:double): double;
var res:double;
begin
  asm
```

```
    FLD f
    FSIN
    FSTP res
  end;
  soproc:=res;
end;
begin
  d:=-pi;
  while (d<=pi) do
  begin
{печать с вызовом ассемблерной функции}
    writeln(d:10:2,'-',soproc(d):10:2);
    d:=d+0.1;
  end;
end.
```

Листинг 3.7.8. Пример использования директивы `asm` и команд сопроцессора в программе на языке C (Visual C++)

```
#include <windows.h>
#include <stdio.h>
double soproc(double f);
void main()
{
  double w=-3.14;
  while(w<=3.14)
  {
    //вывод с использованием ассемблерной процедуры
    printf("%f- %f\n",w,soproc(w));
    w=w+0.1;
  }
  ExitProcess(0);
}

//функция написана на языке ассемблера
double soproc(double f)
{
  double d;
  _asm {
    FLD f
    FSIN
    FSTP d
  }
  return d;
}
```

Пример использования динамической библиотеки

Рассмотрим теперь пример работы с динамической библиотекой, созданной средствами Delphi, из программы на языке ассемблера. Обратим внимание, что никаких принципиальных отличий использования динамических библиотек, написанных на языках высокого уровня, от использования динамических библиотек, написанных на языке ассемблера, нет. И это тоже способ интеграции языка ассемблера с языками высокого уровня.

В листинге 3.7.9 представлен текст программы на языке Object Pascal. Если скомпилировать текст компилятором Delphi, то будет создана динамическая библиотека, которая предоставляет в распоряжение других программ процедуру `setup`. В листинге 3.7.10 содержится программа на языке ассемблера, которая загружает динамическую библиотеку и запускает процедуру `setup`. При запуске программы на экране появляется вначале окно из рис. 3.7.2, а затем окно из рис. 3.7.3.

Листинг 3.7.9. Пример динамической библиотеки, написанной на Delphi

```
library lnk;

uses SysUtils, Classes, Windows;

procedure setup(s1:PChar; s2:PChar); stdcall;
  var s :string;
  begin
    s:='Результирующая строка: '+string(s1)+string(s2);
    MessageBox(0,PAnsiChar(s),'Сообщение из Dll',0);
  end;
//*********************
procedure DLLMain(r:DWORD);
  begin
  end;
  exports setup;
  begin
    DLLProc:=@DLLMain;
    DLLMain(dll_Process_Attach);
    MessageBox(0,'Загрузка динамической библиотеки','Сообщение!',0);
  end.
```

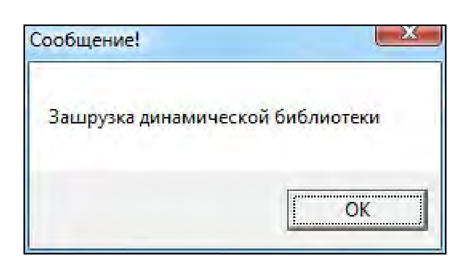


Рис. 3.7.2. Сообщение, появляющееся при успешной загрузке динамической библиотеки из листинга 3.7.9

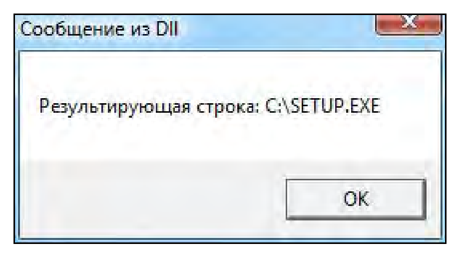


Рис. 3.7.3. Сообщение, появляющееся при успешном вызове процедуры из динамической библиотеки

Листинг 3.7.10. Пример программы на языке ассемблера, осуществляющей вызов динамической библиотеки в листинге 3.7.9

```
;файл get_dll.asm
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
;прототипы внешних процедур
EXTERN  GetProcAddress@8:NEAR
EXTERN  LoadLibraryA@4:NEAR
EXTERN  FreeLibrary@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  MessageBoxA@16:NEAR
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
      TXT    DB 'Ошибка динамической библиотеки',0
      MS     DB 'Сообщение',0
      LIBR   DB 'PROJECT2.DLL',0
      HLIB   DD ?
      PAR1   DB "C:\",0
      PAR2   DB "SETUP.EXE",0
      NAMEPROC DB 'setup',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
; [EBP+10H]   ;резервный параметр
; [EBP+0CH]   ;причина вызова
```

```
; [EBP+8]     ;идентификатор DLL-модуля
START:
;загрузить библиотеку
      PUSH OFFSET LIBR
      CALL LoadLibraryA@4
      CMP  EAX,0
      JE   _ERR
      MOV  HLIB,EAX
;получить адрес
      PUSH OFFSET NAMEPROC
      PUSH HLIB
      CALL GetProcAddress@8
      CMP  EAX,0
      JNE  YES_NAME
;сообщение об ошибке
_ERR:
      PUSH  0
      PUSH  OFFSET MS
      PUSH  OFFSET TXT
      PUSH  0
      CALL  MessageBoxA@16
      JMP  _EXIT
YES_NAME:
      PUSH OFFSET PAR2
      PUSH OFFSET PAR1
      CALL EAX
;закрыть библиотеку
;библиотека автоматически закрывается также при выходе из программы
      PUSH HLIB
      CALL FreeLibrary@4
;выход
_EXIT:
      PUSH 0
      CALL ExitProcess@4
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.7.10:

```
ml /c /coff  get_dll.asm
link /subsystem:windows get_dll.obj
```

А теперь комментарий к листингам 3.7.9 и 3.7.10.

Обратим внимание, что для нас в вызове динамической библиотеки нет ничего нового. При загрузке библиотеки запускается процедура `DLLMain`. После этого программа ищет в загруженной динамической библиотеке нужную

процедуру по ее имени (функция `GetProcAddress`) и, получив адрес этой процедуры, может запустить ее обычной командой `CALL`. При необходимости в процедуру могут быть переданы параметры. Заметим, что для процедуры `setup` объявлено соглашение о вызове, и, следовательно, в ассемблерном модуле мы должны придерживаться этого соглашения.

Использование языка С из программ, написанных на языке ассемблера

Материал, представленный в этом разделе, предоставляет неограниченные возможности программистам, пишущим на языке ассемблера. Вы можете использовать в своих программах всю мощь языка С, в том числе и библиотечные функции этого языка. Собственно, мы уже частично разбирали такую возможность ранее, когда рассматривали простейший калькулятор (см. листинг 3.7.6). Но там главной все же была программа, написанная на языке С. Сейчас мы рассмотрим ситуацию, когда программа на ассемблере использует возможности языка С. Для компоновки программ в данном разделе мы используем программу LINK.EXE из пакета Visual Studio .NET.

Листинг 3.7.11. Программный модуль на языке С, используемый из ассемблерной программы (см. листинг 3.7.12)

```
//программный модуль cc.cpp
#include <windows.h>
#include <stdio.h>

extern "C" void print(char * ,int , int );
extern "C" int subs(int, int);

//данная функция будет вызвана из ассемблерного модуля
void print(char * s,int a, int b)
{
  int c;
  //вызов функции, расположенной в ассемблерном модуле
  c=subs(a,b);
  //вызов библиотечной С-функции
  printf("%s %d",s,c);
  return;
}
```

Трансляция программы из листинга 3.7.11 выполняется так. Для того чтобы получить объектный модуль из модуля, представленного в листинге 3.7.11, следует обратиться к пункту меню **Build | Compile** интегрированной среды Visual Studio .NET.

Листинг 3.7.12. Программа на языке ассемблера, использующая процедуры на языке С

```
;программный модуль cc.asm
.586P
;плоская модель памяти
.MODEL FLAT,C
;-----------------------------------------------
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\user32.lib
EXTERN  CharToOemA@8:NEAR
;сегмент данных
PUBLIC subs
PUBLIC main
EXTERN print:near
DATA SEGMENT
      S DB 'Разность равна',0
DATA ENDS
;сегмент кода
TEXT SEGMENT
main proc
      PUSH OFFSET s
      PUSH OFFSET s
      CALL CharToOemA@8
      MOV  EAX,100
      PUSH EAX
      MOV  EAX,500
      PUSH EAX
      LEA  EAX,S
      PUSH EAX
      CALL print
      ADD  ESP,12
      RET
main endp
subs PROC
      PUSH EBP
      MOV  EBP,ESP
      MOV  EAX,DWORD PTR [EBP+8]
      SUB  EAX,DWORD PTR [EBP+12]
      MOV  ESP,EBP
      POP  EBP
      RET
subs  ENDP
TEXT ENDS
END
```

Для того чтобы правильно транслировать программу, представленную в листинге 3.7.12, нам необходимо:

1. Использовать компоновщик LINK.EXE из пакета Visual Studio .NET. Компоновщик из пакета MASM32, к сожалению, с данной задачей не справится.
2. Скопировать в каталог, где находятся объектные модули, следующие файлы, которые можно найти в пакете Visual Studio .NET: kernel32.lib, libcmt.lib, msobj80.dll, mspdb80.dll, msvcrt.dll, oldname.lib, uuid.lib[1].
3. Наконец, выполнить следующие команды:

```
c:\masm32\bin\ml /c /coff  3-94.asm
"c:\Program Files\Microsoft Visual Studio 8\vc\bin\link.exe" /subsystem:console
3-94.obj cc.obj
```

В результате работы программы на консоль будет выведена строка:

```
Разность равна 400
```

А теперь комментарий к программным модулям из листингов 3.7.11 и 3.7.12.

- Заметим, что в данном примере ведущим является ассемблерный модуль, который вызывает процедуру `print` из модуля на языке C. В свою очередь, из модуля на языке C снова вызывается процедура (`subs`) из ассемблерного модуля. Для удобства в качестве соглашения о вызовах принято соглашение C. В модуле на языке C мы не указываем вид соглашения, т. к. это соглашение действует по умолчанию.
- При трансляции программы, написанной на языке C, компоновщик автоматически использует библиотеку LIBCMT.LIB (в предыдущих версиях пакета Visual C++ эта библиотека называлась LIBC.LIB), которая, кстати, не указана в списке библиотек командной строки LINK.EXE свойств проекта Visual C++. Это очень важная библиотека, в которой содержатся стандартные функции языка C (`printf`, `scanf` и множество других). Для того чтобы эти функции правильно работали (по крайней мере, функции ввода/вывода), необходима начальная инициализация библиотеки. По этой причине программа компонуется так, что выполнение начинается не с функции `main` (`WinMain` для Windows-приложения — параметр компоновщика `/SUBSYSTEM:WINDOWS`), а с функции `mainCRTStartup` (`WinMainCRTStartup` для Windows-приложения), которая находится в библиотеке LIBCMT.LIB. После выполнения процедуры инициализации вызывается функция `main` (или `WinMain` для приложения Windows), и начинает работать код написанной нами программы. Основной вывод, который следует сделать нам: для

[1] Я не помещаю перечисленные файлы на прилагаемый к книге компакт-диск, дабы не нарушать закон об авторских правах и лицензии на программные продукты Microsoft.

того чтобы пользоваться библиотечными функциями языка С, следует обеспечить начальную инициализацию. Нам на помощь приходит программа LINK.EXE из пакета Visual Studio .NET. Оказывается, если в директиве `END` у основного модуля не указывать метку начала выполнения программы (мы обычно указывали метку `START`), то компоновщик автоматически будет искать имя `mainCRTStartup` (или `WinMainCRTStartup`), чтобы сделать стартовой процедуру с этим именем. Нам надо лишь обеспечить наличие такой процедуры, т. е. обеспечить библиотеку LIBCMT.LIB в текущем каталоге. После инициализации будет вызвана процедура `main`, которая как раз имеется в нашем ассемблерном модуле и объявлена как `PUBLIC`.

Наконец, в последнем примере данной главы (листинг 3.7.13) мы продемонстрируем, как из программы на языке ассемблера использовать библиотеки языка С (из пакета Visual Studio .NET).

Листинг 3.7.13. Программа на ассемблере, использующая библиотеку языка С

```
.586P
;плоская модель памяти
.MODEL FLAT,C
;-------------------------------------------------
includelib c:\masm32\lib\user32.lib

INCLUDELIB LIBCMT.LIB
EXTERN  CharToOemA@8:NEAR
PUBLIC main
EXTERN printf:near
;сегмент данных
DATA SEGMENT
      S DB 'Печать числа %d',0
DATA ENDS
;сегмент кода
TEXT SEGMENT
main proc
      PUSH OFFSET s
      PUSH OFFSET s
      CALL CharToOemA@8
      PUSH 100
      LEA  EAX,S
      PUSH EAX
;вызов библиотечной функции
      CALL printf
      ADD  ESP,8
```

```
      RET
main endp
TEXT ENDS
END
```

Для того чтобы выполнить трансляцию программы, представленной в листинге 3.7.13, нам придется соблюсти все те условия, которые были перечислены в соответствующем комментарии к листингу 3.7.12. Команды компиляции программы аналогичны командам из предыдущего примера:

```
c:\masm32\bin\ml /c /coff  3-95.asm
"c:\Program Files\Microsoft Visual Studio 8\vc\bin\link.exe"
/subsystem:console 3-95.obj cc.obj
```

Прокомментирую программу из листинга 3.7.13.

В принципе программа вполне аналогична программе из листинга 3.7.12. Есть, правда, маленький нюанс. Появилась строка `INCLUDELIB LIBCMT.LIB`, которой не было в предыдущем примере. Это объясняется довольно просто: в предыдущем примере ассемблерный код компоновался с кодом на языке С, который как раз и неявно предполагал наличие команды подключения библиотеки LIBCMT.LIB.

Заключая данную главу, отмечу, что мы здесь рассмотрели далеко не все возможные случаи и проблемы стыковки ассемблера с языками высокого уровня. Разобрав, однако, предложенные примеры, вы сможете самостоятельно решить все подобные задачи.

Глава 3.8

Программирование сервисов

Сервис — что это за птица такая? Это что, программы такие особые? И для чего их использовать? Наберитесь терпения — обо всем узнаете в свое время, но в данной главе.

Основные понятия и функции управления

Сервис или служба — это особое приложение. Корпорация Microsoft рекомендует реализовывать серверные приложения именно в виде сервисов. В самой операционной системе Windows множество функций выполняют именно сервисы. Типичным примером такой службы является служба Plug and Play. Эта служба отслеживает все изменения в аппаратной конфигурации компьютера и управляет установкой и настройкой устройств. Еще одной службой, которая работает у меня на компьютере и благодаря которой я могу использовать в приложениях обращения к SQL Server, является MSSQLSERVER. А в общем-то, служб работает множество. Чтобы убедиться в этом, достаточно обратиться к специальному приложению — SCP (Service Control Program, программа управления сервисами). Эта программа позволяет управлять службами вашего компьютера в диалоговом режиме. Запустить ее можно из папки **Администрирование**. На рис. 3.8.1 вы можете видеть окно этой программы.

На рис. 3.8.2 представлено окно управления сервисом. Обращаю ваше внимание на четыре кнопки: **Запустить**, **Остановить**, **Приостановить**, **Продолжить**. С помощью этих кнопок можно выполнить следующие четыре операции над выбранной службой:

- запуск службы;
- останов службы. Останов в частности необходим для удаления службы;

Службы

Консоль Действие Вид Справка

Службы (локальны

Службы (локальные)

Локатор удаленного вызова процедур (RPC)

Запустить службу

Описание:
Управляет базой данных службы имен RPC.

Имя	Описание	Состояние	Тип запуска	Вх
Диспетчер печати	Загрузка файлов в память, что...	Работает	Авто	Ло
Диспетчер подключений удаленного доступа	Управляет подключениями уда...	Работает	Вручную	Ло
Диспетчер сеансов диспетчера окон рабочего стола	Обеспечивает запуск и обслуж...	Работает	Авто	Ло
Диспетчер удостоверения сетевых участников	Предоставляет службу иденти...		Вручную	Lo
Диспетчер учетных записей безопасности	Запуск этой службы служит дл...	Работает	Авто	Ло
Доступ к HID-устройствам	Обеспечивает универсальный ...		Вручную	Ло
Журнал событий Windows	Эта служба управляет события...	Работает	Авто	Lo
Журналы и оповещения производительности	Служба журналов производит...		Вручную	Lo
Защитник Windows	Проверка компьютера на нали...	Работает	Авто	Ло
Защищенное хранилище	Обеспечивает защищенное хр...		Вручную	Ло
Изоляция ключей CNG	Служба изоляции ключей CNG...		Вручную	Ло
Инструментарий управления Windows	Предоставляет общий интерф...	Работает	Авто	Ло
Информация о совместимости приложений	Обработка запросов на провер...	Работает	Авто	Ло
Клиент групповой политики	Данная служба ответственна за...	Работает	Авто	Ло
Клиент отслеживания изменившихся связей	Поддерживает связи NTFS-фай...	Работает	Авто	Ло
Координатор распределенных транзакций	Координация транзакций, охва...		Вручную	Ne
Кэш шрифтов Windows Presentation Foundation 3.0.0.0	Оптимизирует производительн...		Вручную	Lo
Лицензирование программного обеспечения	Разрешает загрузку, установку ...	Работает	Авто	Ne
Ловушка SNMP	Принимает сообщения перехв...		Вручную	Lo
Локатор удаленного вызова процедур (RPC)	Управляет базой данных служб...		Вручную	Ne
Маршрутизация и удаленный доступ	Предлагает услуги маршрутиз...		Отключена	Ло
Модули ключей IPsec для обмена ключами в Интернет...	Служба IKEEXT содержит моду...	Работает	Авто	Ло
Модуль запуска процессов DCOM-сервера	Обеспечивает запуск для служ...	Работает	Авто	Ло
Модуль поддержки NetBIOS через TCP/IP	Осуществляет поддержку NetB...	Работает	Авто	Lo
Настройка служб терминалов	Служба настройки служб терм...		Вручную	Ло
Немедленные подключения Windows - регистратор нас...	Выступает в роли регистратор...		Вручную	Lo
Обнаружение SSDP	Обнаруживает сетевые устрой...	Работает	Вручную	Lo

Расширенный Стандартный

Рис. 3.8.1. Вид диалогового окна программы управления службами Microsoft Windows Vista

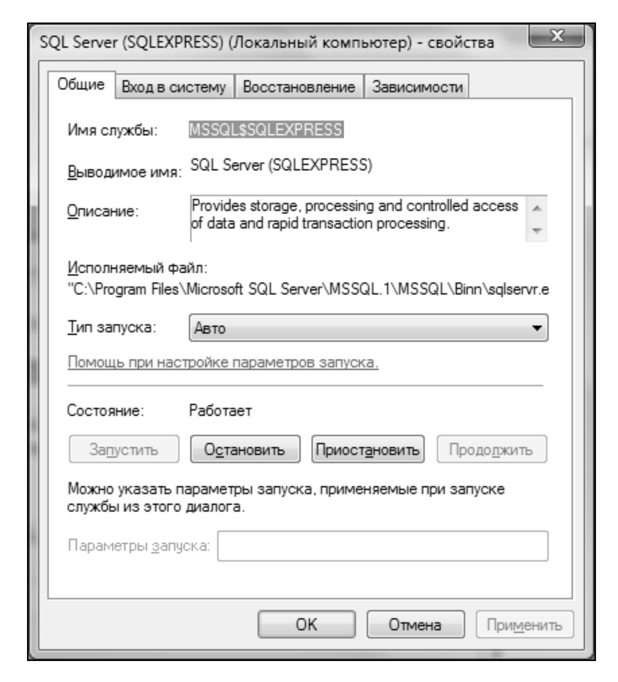


Рис. 3.8.2. Окно управления сервисом

- приостановить службу;
- продолжить приостановленную службу.

Можно изменить тип запуска службы с помощью раскрывающегося списка **Тип запуска**:

- **Авто** — автоматический запуск сервиса диспетчером управления сервисами. Сервис запускается при старте операционной системы, еще до того, как в системе будет зарегистрирован какой-либо пользователь (!);
- **Вручную** — предполагается, что сервис будет запущен администратором, во время работы системы. Запуск может осуществиться как с помощью кнопки **Пуск**, так и программным путем, с помощью API-функции `StartService`;
- **Отключено** — заблокированная служба.

На вкладке **Вход в систему** администратор может установить, под каким именем будет входить в систему сервис. По умолчанию это имя LocalSystem. Под данным именем сервис может выполнять в системе практически любые действия. Но вы можете указать на данной вкладке, что служба должна входить под конкретным именем и паролем пользователя.

Вкладка **Восстановление** позволяет определить, какие действия должен совершить менеджер сервисов, при аварийном завершении сервиса. Можно определить действия на первый, второй и последующие сбои.

Вкладка **Зависимости** отображает все сервисы, зависящие от данного сервиса, и все сервисы, от которых зависит данный сервис.

Сервисы Windows работают под управлением диспетчера управления службами (SCM, Service Control Manager). Весь интерфейс с сервисом будет осуществляться через данную программу. Эта программа называется services.exe, которая хранится в системном каталоге (System32) и запускается автоматически вместе с операционной системой, оставаясь в активном состоянии до конца работы операционной системы.

Список служб можно найти и в реестре. Данные обо всех сервисах можно найти в системном реестре по адресу:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services
```

На рис. 3.8.3 представлено окно программы regedt32.exe (программа расположена в подкаталоге System32 каталога Windows) как раз со списком сервисов.

> ***Замечание***
>
> Избави вас боже менять что-либо в настройках или удалять не ваш сервис, не зная его назначения и связей с другими сервисами, из реестра. Это может закончиться для вашей системы печально. Но если вы создаете свой сервис, то такое удаление, однако, в некоторых случаях окажется единственной возможностью отладить ваше приложение с наименьшими издержками.

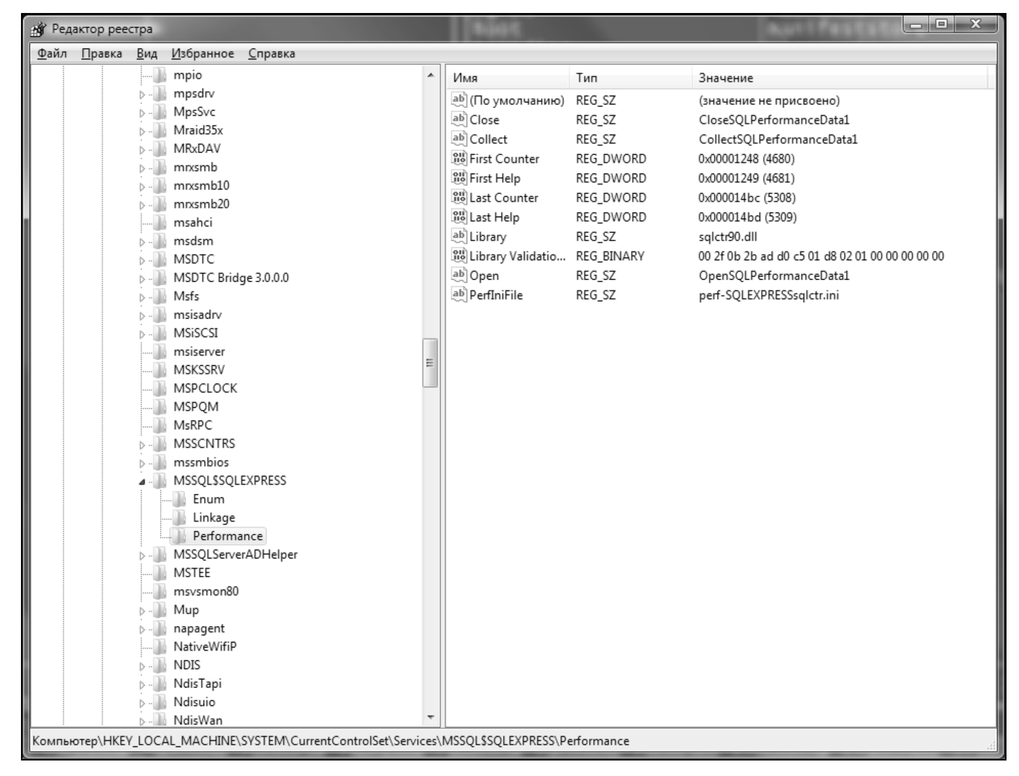


Рис. 3.8.3. Список сервисов локального компьютера из окна программы regedt32.exe

Структура сервисов

Обратимся теперь к структуре сервиса, а также обсудим некоторые API-функции, которые использовать для управления сервисами.

С точки зрения операционной системы сервис имеет структуру обычного загружаемого модуля. Однако запускается он не обычным образом и имеет три четко обозначенные части.

- Основная функция процесса. Ее начало указывает обычный стартовый адрес программы. В наших приложениях этот стартовый адрес я всегда обозначаю меткой `START`. В простейшем случае здесь должен стоять только вызов функции `StartServiceCtrlDispatcher`. В задачу данной функции входит регистрация сервисов и запуск диспетчера управления сервисами. Я не оговорился: в одной программе может быть несколько сервисов, фактически представляющих обычные функции. Можно назвать это логическими сервисами. Каждая такая функция будет запускаться диспетчером в собствен-

ном потоке. Единственным параметром функции `StartServiceCtrlDispatcher` является адрес массива, состоящего из пар: имя и точка входа (адрес) службы. В конце массива должно стоять два нуля (см. программу из листинга 3.8.1).

- Процедуры или логические службы, которые мы указываем с помощью функции `StartServiceCtrlDispatcher`, должны содержать три компонента:
  - вызов функции `RegisterServiceCtrlHandler`, с помощью которой регистрируется процедура-обработчик команд для данного логического сервиса;
  - вызов функции `SetServiceStatus`, которая устанавливает статус службы — "служба запущена";
  - выполнение действия, для которых предназначена данная логическая служба. Как мы уже говорили, некоторые службы работают в течение всего периода работы операционной системы, тогда как другие службы запускаются на короткое время. Логическая служба может, в свою очередь, запускать произвольное количество потоков. По окончанию выполнения действия логическая служба должна опять с помощью функции `SetServiceStatus` сообщить о своем состоянии — "служба остановлена".
- Обработчик команд (handler). Диспетчер сервисов взаимодействует с обработчиком, вынуждая, например, остановить службу или выдать сообщение о ее состоянии.

Да, это практически вся структура сервиса. Нам теперь предстоит описать упомянутые или еще не упомянутые функции API, используемые в работе с сервисами.

Функцию `StartServiceCtrlDispatcher` мы уже фактически описали, и в следующем разделе вы увидите, как она используется.

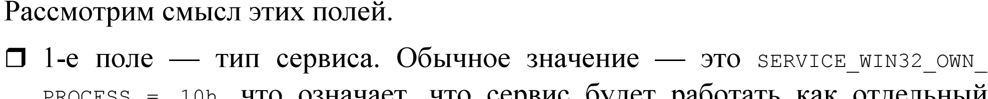
Функция `RegisterServiceCtrlHandler` — регистрация процедуры обработчика команд. Функция возвращает дескриптор. Параметры функции:

- 1-й параметр — адрес имени логического сервиса;
- 2-й параметр — адрес процедуры обработки команд.

Функция `SetServiceStatus` — устанавливает статус сервиса. Параметры функции:

- 1-й параметр — дескриптор процедуры обработки команд;
- 2-й параметр — указатель на структуру, состоящую из семи 32-битных полей.

Рассмотрим смысл этих полей.



- 1-е поле — тип сервиса. Обычное значение — это `SERVICE_WIN32_OWN_PROCESS = 10h`, что означает, что сервис будет работать как отдельный процесс.
- 2-е поле определяет текущее состояние сервиса. Например, `SERVICE_RUNNING = 4h`, что означает, что сервис запущен и работает.
- 3-е поле определяет, какие команды будет обрабатывать сервис. Существует несколько констант, которые объединяются с помощью операции "ИЛИ" (см. листинг 3.8.1).
- 4-е поле — код, определяющей, например, останов сервиса. Обычно поле полагают равным 0.
- 5-е поле — код специфической ошибки. Используется, если предыдущее поле не равно нулю.
- 6-е поле должно периодически увеличиваться при выполнении длительных операций. В остальных случаях оно должно быть равно нулю.
- 7-е поле — здесь следует указывать оценочное время выполнения длительных операций. Если за указанное время не изменится значение второго поля или предыдущего поля, то система будет считать, что произошла ошибка.

Мы разобрали основные функции, которые обязательно должны присутствовать в программе-сервисе. В простейшем случае обработчик команд также будет содержать только функцию `SetServiceStatus`, с помощью которой будет устанавливаться статус сервиса, в зависимости от поступаемой команды. Конкретная структура всех трех составляющих частей сервиса будет приведена в программе из листинга 3.8.1.

Приведенных функций API, однако, не достаточно, чтобы заставить сервис работать. Для того чтобы это сделать, сервис должен быть предварительно помещен в базу сервисов, а точнее, в соответствующий раздел реестра. Только после этого он может быть запущен. Довольно часто для выполнения загрузки сервиса в базу используется отдельная программа. Далее мы поступаем именно так. Однако многие программы-сервисы устроены так, что в зависимости от параметров командной строки выполняются те или иные операции над сервисом.

Итак, рассмотрим вопрос помещения сервиса в сервисную базу. Для этого база должна быть предварительно открыта.

Функция `OpenSCManager` открывает базу сервисов. Функция имеет несколько параметров:

- 1-й параметр — имя (адрес имени) рабочей станции в сети, на которой вы хотите открыть базу. Если база открывается на локальном компьютере, то этот параметр полагается равным 0;
- 2-й параметр — адрес имени базы данных сервисов. Для открытия базы данных по умолчанию данный параметр также следует положить равным нулю;
- 3-й параметр определяет нужный тип доступа. Обычно для полного доступа используют константу `SC_MANAGER_ALL_ACCESS = 0F003Fh`.

При успешном выполнении функция возвращает дескриптор базы данных. Если база данных была открыта успешно, то в конце ее следует закрыть. Для этого нужно использовать функцию `CloseServiceHandle`, единственным параметром которой является дескриптор базы сервисов.

Для помещения сервиса в сервисную базу используется API-функция `CreateService`. Функция имеет 13 параметров! Вот они:

- 1-й параметр — дескриптор сервисной базы данных (см. `OpenSCManager`);
- 2-й параметр — адрес строки, содержащей имя одной из логических служб, по которому в дальнейшем будет возможно обращение к этой службе. Можно сделать, таким образом, вывод, что для каждой логической службы придется вызывать функцию `CreateService`. Под этим именем служба появится в системном реестре. По этому имени к сервису можно обращаться программно;
- 3-й параметр определяет так называемое отображаемое имя;
- 4-й параметр — возможный тип доступа к сервису. В моем примере используется `SERVICE_ALL_ACCESS = 0F01FFH` (т. е. полный доступ);
- 5-й параметр — тип сервиса. Я об этом уже говорил. В примере используется константа `SERVICE_WIN32_OWN_PROCESS = 10H`;
- 6-й параметр — тип старта службы. Я в примере использую константу `SERVICE_DEMAND_START = 3`, т. е. запуск по требованию. Можно использовать и другие константы, например `SERVICE_BOOT_START = 0`, т. е. запуск сервиса вместе с запуском системы;
- 7-й параметр определяет уровень реакции на ошибку. Стандартное значение `SERVICE_ERROR_NORMAL = 1`;
- 8-й параметр — с помощью данного параметра указывается на строку, содержащую имя программы-сервиса. Имя должно указывать и путь

(например, так: "D:\masm32\BIN\mt.exe"). Расширение exe указывать не обязательно;

- 9-й параметр — в операционной системе имеется несколько групп служб. Группы служб загружаются в определенном порядке. Так что, если для вас важно, после каких сервисов будет загружен ваш сервис, укажите здесь адрес имени группы, куда будет входить ваш сервис. Названия всех групп имеются в документации. Если для вас порядок загрузки не важен, укажите в качестве данного параметра 0;
- 10-й параметр — данный параметр следует полагать всегда равным нулю, т. к. он используется только для сервисов-драйверов (*см. главу 4.6* о драйверах режима ядра);
- 11-й параметр определяет сервисы и группы сервисов, от которых зависит ваш сервис. Он должен указывать на массив, состоящий из имен сервисов и групп сервисов. Массив должен заканчиваться двойным нулем;
- 12-й параметр указывает на имя учетной записи, с которой должна запускаться служба. Обычно параметр полагают равным нулю, так как предполагается, что служба запускается под именем LocalSystem;
- 13-параметр — пароль учетной записи (см. параметр 12). Если 12-й параметр положен равным 0, то и этот параметр также должен быть равным нулю.

Перейдем теперь к вопросу о программном запуске сервиса, который уже зарегистрирован ранее с помощью функции `CreateService`. Порядок здесь такой:

1. Открыть базу сервисов. Для этого используется описанная ранее функция `OpenSCManager`.
2. Открыть сам сервис с помощью функции `OpenService`.
3. Стартовать сервис с помощью функции `StartService`.
4. Закрыть базу сервисов.

Приступим к описанию еще незнакомых вам функций.

Функция `OpenService` имеет три параметра:

- 1-й параметр — дескриптор базы сервисов;
- 2-й параметр определяет имя открываемого сервиса. Это имя соответствует имени, под которым сервис помещен в системный реестр;
- 3-й параметр определяет тип доступа к сервису. Обычно указывается `SC_MANAGER_ALL_ACCESS` (для удаления можно указать константу `DELETE`).

Функция `StartService` так же, как и предыдущая, имеет три параметра:

- 1-й параметр — дескриптор, возвращаемый функцией `OpenService`;
- 2-й и 3-й параметры определяют параметры, передаваемые в службу. Обычно эту возможность не используют и полагают оба значения равными нулю.

Наконец, разберем еще один важный момент. Как программно удалить сервис из базы сервисов, т. е. из системного реестра? Для этого следует выполнить такие действия:

1. Открыть базу сервисов.
2. Открыть конкретный сервис.
3. Послать сервису команду останова (используется функция `ControlService`) на случай, если он запущен. Если он не запущен, то ничего страшного не произойдет.
4. Удалить сервис с помощью API-функции `DeleteService`.
5. Закрыть базу сервисов.

Опишем еще некоторые полезные функции.

Функция `ControlService`. Имеет три параметра:

- 1-й параметр — дескриптор сервиса;
- 2-й параметр — команда, посылаемая сервису, которая будет передана обработчику. Для останова службы мы посылаем команду `SERVICE_CONTROL_STOP = 1`;
- 3-й параметр — адрес строки, содержащей имя сервиса (внутреннее).

Функция `DeleteService` имеет всего один параметр — дескриптор открытого ранее сервиса.

Пример сервиса

Ну вот, и до самого интересного добрались, до примеров то есть. Чтобы было легче во всем разобраться, я выделил четыре подзадачи: саму программу-сервис, установку сервиса в базу, запуск сервиса, останов и удаление сервиса из базы. Программу serv.exe (листинг 3.8.1) нельзя запустить обычным образом. Попробуйте, ни к чему это не приведет. Структура ее такова, что она запускается, т. е. помещается вначале в базу сервисов особым образом с помощью программы setserv.exe (листинг 3.8.2). Запустить (на выполнение) сервис можно с помощью программы stserv.exe (листинг 3.8.3). Это же действие можно осуществить и с помощью стандартной консоли **Службы**

в стандартном окне **Администрирование**. Наконец, удаляется наша служба вне зависимости от того, выполняется она или нет, с помощью программы delserv.exe (листинг 3.8.4).

Листинг 3.8.1. Пример простейшей программы-сервиса (serv.exe)

```
.586P
;плоская модель
.MODEL FLAT, stdcall
;константы
SERVICE_CONTROL_STOP          equ 1h
SERVICE_CONTROL_SHUTDOWN      equ 5h
SERVICE_CONTROL_INTERROGATE   equ 4h
SERVICE_CONTROL_CONTINUE      equ 3h
SERVICE_START_PENDING         equ 2h
ERROR_SERVICE_SPECIFIC_ERROR  equ 1066
SERVICE_RUNNING               equ 4h
MB_SERVICE_NOTIFICATION       equ 200000h

CRST EQU SERVICE_CONTROL_STOP OR \
         SERVICE_CONTROL_SHUTDOWN OR \
         SERVICE_CONTROL_CONTINUE
SERVICE_WIN32_OWN_PROCESS     equ 00000010h
;прототипы внешних процедур
EXTERN  Sleep@4:NEAR
EXTERN  SetServiceStatus@8:NEAR
EXTERN  RegisterServiceCtrlHandlerA@8:NEAR
EXTERN  StartServiceCtrlDispatcherA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  MessageBoxA@16:NEAR

;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\advapi32.lib
;-----------------------------------------------

SSTATUS STRUC
      STYPE    DD ?
      SSTATE   DD ?
      SACCEPT  DD ?
      SEXCODE  DD ?
      SEXSCOD  DD ?
      SCHEKPO  DD ?
```

```
      SWAITHI  DD ?
SSTATUS ENDS

;сегмент данных
_DATA SEGMENT
      SNAME   DB  "MyService",0
      DTS     DD  OFFSET SNAME,OFFSET WINSERV,0,0
      SRS     SSTATUS <0>
      H1      DD  ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;регистрация сервиса
      PUSH OFFSET DTS
      CALL StartServiceCtrlDispatcherA@4
;выход происходит по завершению всех служб
      PUSH 0
      CALL ExitProcess@4
;сам сервис
WINSERV PROC
;заполнить структуру
      MOV SRS.STYPE,SERVICE_WIN32_OWN_PROCESS
      MOV SRS.SSTATE,SERVICE_RUNNING ;SERVICE_START_PENDING
      MOV SRS.SACCEPT,CRST
      MOV SRS.SEXCODE,0              ;ERROR_SERVICE_SPECIFIC_ERROR
      MOV SRS.SEXSCOD,0
      MOV SRS.SCHEKPO,0
      MOV SRS.SWAITHI,1
;зарегистрировать функцию обработки команд
      PUSH OFFSET HANDLER
      PUSH OFFSET SNAME
      CALL RegisterServiceCtrlHandlerA@8
;установить статус
      MOV  H1,EAX
      PUSH OFFSET SRS
      PUSH H1
      CALL SetServiceStatus@8
;здесь запускается что-то, что составляет
;основную функцию данного сервиса
      PUSH 200000
      CALL Sleep@4
;устанавливаем статус
      MOV SRS.SSTATE,SERVICE_CONTROL_STOP
      PUSH OFFSET SRS
```

```
      PUSH H1
      CALL SetServiceStatus@8
      RET  8
WINSERV ENDP
;обработчик прерываний
; [EBP+08H] - единственный параметр
HANDLER PROC
      PUSH EBP
      MOV  EBP, ESP
      INC  SRS.SCHEKPO
      MOV  EAX, DWORD PTR [EBP+08H]
      CMP  EAX,SERVICE_CONTROL_STOP
      JNZ  NO_STOP
      MOV  SRS.SSTATE,SERVICE_CONTROL_STOP
      JMP _SET
NO_STOP:
      CMP  EAX,SERVICE_CONTROL_SHUTDOWN
      JNZ  NO_SHUTDOWN
      MOV  SRS.SSTATE,SERVICE_CONTROL_STOP
      JMP _SET
NO_SHUTDOWN:
      CMP  EAX,SERVICE_CONTROL_CONTINUE
      JNZ  NO_CONTINUE
      MOV  SRS.SSTATE,SERVICE_CONTROL_CONTINUE
      JMP _SET
NO_CONTINUE:
      CMP  EAX,SERVICE_CONTROL_INTERROGATE
;установить состояние сервиса
_SET:
      PUSH OFFSET SRS
      PUSH H1
      CALL SetServiceStatus@8
;сообщение, вставляемое для отладки
;константа MB_SERVICE_NOTIFICATION обязательна
      PUSH 0 OR MB_SERVICE_NOTIFICATION
      PUSH OFFSET SNAME
      PUSH OFFSET SNAME
      PUSH 0
      CALL MessageBoxA@16
;------------------------------
      MOV  ESP, EBP
      POP  EBP
      RET  4
HANDLER ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.8.1:

```
ML /c /coff  serv.asm
LINK /SUBSYSTEM:CONSOLE serv.obj
```

Комментарий к программе из листинга 3.8.1.

- Обратите внимание на структуру программы. В ней, как раньше говорилось, присутствуют: основная функция (метка `START`), логический сервис (он здесь один) — `WINSERV`, обработчик команд — `HANDLER`.
- Чтобы не мудрствовать лукаво, мы в качестве исполняемого процесса используем функцию `Sleep` с большой задержкой. Когда мы запускаем программу stserv, то запускается в конечном итоге именно эта функция. В принципе сервис не обязательно останавливать "силой", по окончании работы функции `Sleep` он сам остановится.
- В конце функции `WINSERV` стоит `RET 8`. Это значит, что в функцию посылаются два параметра, которые мы, впрочем, не обрабатываем (см. описание `StartService`).
- В структуре процедуры `HANDLER` нет ничего сложного для программирующего на ассемблере. В сущности, данный обработчик нам нужен только для обработки команды `SERVICE_CONTROL_STOP`.
- С чисто отладочными целями мы поместили в обработчике команд функцию `MessageBox`. Это довольно интересный момент. Сообщение должно появляться относительно конкретного рабочего стола. По этой причине мы используем константу `MB_SERVICE_NOTIFICATION`, предназначенную как раз для вывода сообщений от сервиса. Сообщение появится, даже если не было регистрации пользователя на компьютере.
- Команда `RET 4` в конце обработчика означает, что для обработчика при вызове поступает только один параметр — команда для сервиса.

Листинг 3.8.2. Пример программы, устанавливающей сервис (setserv.exe)

```
.586P
;плоская модель
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE         equ -11
SC_MANAGER_ALL_ACCESS     equ 0F003Fh
SERVICE_ALL_ACCESS        equ 0F01FFH
SERVICE_WIN32_OWN_PROCESS equ 00000010h
SERVICE_DEMAND_START      equ 00000003h
SERVICE_ERROR_NORMAL      equ 00000001h
;прототипы внешних процедур
```

```
EXTERN  CreateServiceA@52:NEAR
EXTERN  CloseServiceHandle@4:NEAR
EXTERN  OpenSCManagerA@12:NEAR
EXTERN  wsprintfA:NEAR
EXTERN  GetLastError@0:NEAR
EXTERN  StartServiceCtrlDispatcherA@4:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  GetStdHandle@4:NEAR

;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\advapi32.lib
;-----------------------------------------------

;сегмент данных
_DATA SEGMENT
      H1      DD ?
      H2      DD ?
      ALIGN   4
      SNAME1  DB  "MyService",0
      ALIGN   4
;здесь правильный путь к службе
      NM      DB "c:\masm32\primers\gl-3\3-96\serv.exe",0
      LENS    DD 0
      HANDL   DD 0
      BUF1    DB 512 DUP(0)
      ERRS    DB "Error %u ",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;определить дескриптор консоли вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;открыть базу служб
      PUSH SC_MANAGER_ALL_ACCESS
      PUSH 0
      PUSH 0
      CALL OpenSCManagerA@12
      CMP  EAX,0
      JNZ  NO_ERR
      CALL ERROB
```

```
      JMP  EXI
NO_ERR:
      MOV  H1,EAX
;идентификатор получен, создаем сервис
      PUSH 0
      PUSH 0
      PUSH 0
      PUSH 0
      PUSH 0
      PUSH OFFSET NM
      PUSH SERVICE_ERROR_NORMAL
      PUSH SERVICE_DEMAND_START
      PUSH SERVICE_WIN32_OWN_PROCESS
      PUSH SERVICE_ALL_ACCESS
      PUSH OFFSET SNAME1
      PUSH OFFSET SNAME1
      PUSH H1
      CALL CreateServiceA@52
      CMP  EAX,0
      JNZ  CLOS
      MOV  H2,EAX
      CALL ERROB
      JMP  CLOS1
;здесь блок обработки ошибок
ERROB:
      CALL GetLastError@0
      PUSH EAX
      PUSH OFFSET ERRS
      PUSH OFFSET BUF1
      CALL wsprintfA
      ADD  ESP,12
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
      RET
CLOS1:
;закрыть сервис
      PUSH H2
      CALL CloseServiceHandle@4
CLOS:
;закрыть базу сервисов
      PUSH H1
      CALL CloseServiceHandle@4
EXI:
;выход происходит по завершению всех служб
      PUSH 0
```

```
      CALL ExitProcess@4
;вывести строку (в конце перевод строки)
;EAX - на начало строки
;EDI - с переводом строки или без
WRITE   PROC
;получить длину параметра
      PUSH  EAX
      PUSH  EAX
      CALL  lstrlenA@4
      MOV   ESI,EAX
      POP   EBX
      CMP   EDI,1
      JNE   NO_ENT
;в конце - перевод строки
      MOV  BYTE PTR [EBX+ESI],13
      MOV  BYTE PTR [EBX+ESI+1],10
      MOV  BYTE PTR [EBX+ESI+2],0
      ADD  EAX,2
NO_ENT:
;вывод строки
      PUSH 0
      PUSH OFFSET LENS
      PUSH EAX
      PUSH EBX
      PUSH HANDL
      CALL WriteConsoleA@20
      RET
WRITE  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.8.2:

```
ML /c /coff /DMASM setserv.asm
LINK /SUBSYSTEM:CONSOLE setserv.obj
```

Комментарий к программе из листинга 3.8.2 (setserv.exe).

- Данная программа заносит программу-сервис serv.exe в сервисную базу, т. е. в системный реестр. Для этого используется API-функция `CreateService`.
- Обратите внимание, что здесь мы обрабатываем возможные ошибки при использовании этой функции и в случае их возникновения выводим на консоль номер ошибки.
- Путь к каталогу, где находится служба, должен быть абсолютным, а не относительным. В нашем примере он равен "c:\masm32\primers\gl-3\3-96\serv.exe", измените его для своей ситуации. В противном случае (если

путь относителен) службу не удастся запустить (программа stserv.exe), хотя она и будет установлена программой setserv.exe.

Листинг 3.8.3. Пример программы, запускающей сервис (stserv.exe)

```
.586P
;плоская модель
.MODEL FLAT, stdcall
;константы
DELETE                   equ 10000h
STD_OUTPUT_HANDLE        equ -11
SC_MANAGER_ALL_ACCESS    equ 0F003Fh
SERVICE_ALL_ACCESS       equ 0F01FFH
SERVICE_WIN32_OWN_PROCESS equ 00000010h
SERVICE_DEMAND_START     equ 00000003h
SERVICE_ERROR_NORMAL     equ 00000001h
SERVICE_CONTROL_STOP     equ 1h
;прототипы внешних процедур
EXTERN  StartServiceA@12:NEAR
EXTERN  OpenServiceA@12:NEAR
EXTERN  CloseServiceHandle@4:NEAR
EXTERN  OpenSCManagerA@12:NEAR
EXTERN  wsprintfA:NEAR
EXTERN  GetLastError@0:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  GetStdHandle@4:NEAR

;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\advapi32.lib
;-------------------------------------------------

SSTATUS STRUC
      STYPE    DD ?
      SSTATE   DD ?
      SACCEPT  DD ?
      SEXCODE  DD ?
      SEXSCOD  DD ?
      SCHEKPO  DD ?
      SWAITHI  DD ?
SSTATUS ENDS

;сегмент данных
```

```
_DATA SEGMENT
      SRS     SSTATUS <?>
      H1      DD ?
      H2      DD ?
      ALIGN   4
      SNAME1  DB  "MyService",0
      ALIGN   4
      LENS    DD 0
      HANDL   DD 0
      BUF1    DB 512 DUP(0)
      ERRS    DB "Error %u ",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;------------------------
      PUSH SC_MANAGER_ALL_ACCESS
      PUSH 0
      PUSH 0
      CALL OpenSCManagerA@12
      CMP  EAX,0
      JNZ  NO_ERR1
      CALL ERROB
      JMP  EXI
NO_ERR1:
      MOV  H1,EAX
;идентификатор получен
      PUSH SC_MANAGER_ALL_ACCESS ;DELETE
      PUSH OFFSET SNAME1
      PUSH H1
      CALL OpenServiceA@12
      CMP  EAX,0
      JNZ  NO_ERR2
      CALL ERROB
      JMP  CLOS
NO_ERR2:
      MOV  H2,EAX
;идентификатор сервиса получен
;даем команду старта сервиса
      PUSH 0
      PUSH 0
      PUSH H2
      CALL StartServiceA@12
      CMP  EAX,0
```

```
      JNZ  CLOS1
      CALL ERROB
      JMP  CLOS1
;блок обработки ошибок
ERROB:
      CALL GetLastError@0
      PUSH EAX
      PUSH OFFSET ERRS
      PUSH OFFSET BUF1
      CALL wsprintfA
      ADD  ESP,12
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
      RET
CLOS1:
;закрыть сервис
      PUSH H2
      CALL CloseServiceHandle@4
;закрыть базу сервисов
CLOS:
      PUSH H1
      CALL CloseServiceHandle@4
EXI:
;выход
      PUSH 0
      CALL ExitProcess@4
;вывести строку (в конце перевод строки)
;EAX - на начало строки
;EDI - с переводом строки или без
WRITE   PROC
;получить длину параметра
      PUSH  EAX
      PUSH  EAX
      CALL  lstrlenA@4
      MOV   ESI,EAX
      POP   EBX
      CMP   EDI,1
      JNE   NO_ENT
;в конце - перевод строки
      MOV  BYTE PTR [EBX+ESI],13
      MOV  BYTE PTR [EBX+ESI+1],10
      MOV  BYTE PTR [EBX+ESI+2],0
      ADD  EAX,2
NO_ENT:
;вывод строки
      PUSH 0
```

```
      PUSH OFFSET LENS
      PUSH EAX
      PUSH EBX
      PUSH HANDL
      CALL WriteConsoleA@20
      RET
WRITE  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.8.3:

```
ML /c /coff /DMASM stserv.asm
LINK /SUBSYSTEM:CONSOLE stserv.obj
```

Программа из листинга 3.8.3 (stserv.exe) запускает службу с заданным именем `MyService` на выполнение. Запуск осуществляется с помощью API-функции `StartService`.

Листинг 3.8.4. Пример программы, удаляющей сервис (delserv.exe)

```
.586P
;плоская модель
.MODEL FLAT, stdcall
;константы
DELETE                    equ 10000h
STD_OUTPUT_HANDLE         equ -11
SC_MANAGER_ALL_ACCESS     equ 0F003Fh
SERVICE_ALL_ACCESS        equ 0F01FFH
SERVICE_WIN32_OWN_PROCESS equ 00000010h
SERVICE_DEMAND_START      equ 00000003h
SERVICE_ERROR_NORMAL      equ 00000001h
SERVICE_CONTROL_STOP      equ 1h
;прототипы внешних процедур
EXTERN  ControlService@12:NEAR
EXTERN  DeleteService@4:NEAR
EXTERN  OpenServiceA@12:NEAR
EXTERN  CloseServiceHandle@4:NEAR
EXTERN  OpenSCManagerA@12:NEAR
EXTERN  wsprintfA:NEAR
EXTERN  GetLastError@0:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  GetStdHandle@4:NEAR

;директивы компоновщику для подключения библиотек
```

```
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\advapi32.lib
;-----------------------------------------------

SSTATUS STRUC
      STYPE    DD ?
      SSTATE   DD ?
      SACCEPT  DD ?
      SEXCODE  DD ?
      SEXSCOD  DD ?
      SCHEKPO  DD ?
      SWAITHI  DD ?
SSTATUS ENDS

;сегмент данных
_DATA SEGMENT
      SRS     SSTATUS <?>
      H1      DD ?
      H2      DD ?
      ALIGN   4
      SNAME1  DB  "MyService",0
      ALIGN   4
      LENS    DD 0
      HANDL   DD 0
      BUF1    DB 512 DUP(0)
      ERRS    DB "Error %u ",0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;открыть базу служб
      PUSH SC_MANAGER_ALL_ACCESS
      PUSH 0
      PUSH 0
      CALL OpenSCManagerA@12
      CMP  EAX,0
      JNZ  Z1
      CALL ERROB
      JMP  EXI
Z1:
      MOV  H1,EAX
;идентификатор получен, открыть сервис
      PUSH SC_MANAGER_ALL_ACCESS ;DELETE
```

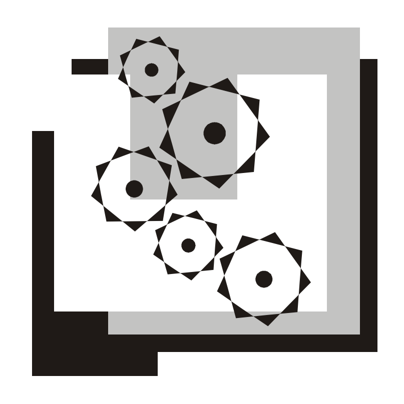
```
      PUSH OFFSET SNAME1
      PUSH H1
      CALL OpenServiceA@12
      CMP  EAX,0
      JNZ  Z2
      CALL ERROB
      JMP  CLOS
Z2:
      MOV  H2,EAX
;дать команду остановки
      PUSH OFFSET SRS
      PUSH SERVICE_CONTROL_STOP
      PUSH H2
      CALL ControlService@12
      CMP  EAX,0
      JNZ  Z3
      CALL ERROB
Z3:
;удалить сервис
      PUSH H2
      CALL DeleteService@4
      CMP  EAX,0
      JNZ  CLOS
      CALL ERROB
      JMP  CLOS1
;блок обработки ошибок
ERROB:
      CALL GetLastError@0
      PUSH EAX
      PUSH OFFSET ERRS
      PUSH OFFSET BUF1
      CALL wsprintfA
      ADD  ESP,12
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
      RET
CLOS1:
;закрыть сервис
      PUSH H2
      CALL CloseServiceHandle@4
CLOS:
;закрыть базу сервисов
      PUSH H1
      CALL CloseServiceHandle@4
EXI:
;выход происходит по завершению всех служб
      PUSH 0
```

```
      CALL ExitProcess@4
;вывести строку (в конце перевод строки)
;EAX - на начало строки
;EDI - с переводом строки или без
WRITE  PROC
;получить длину параметра
      PUSH  EAX
      PUSH  EAX
      CALL  lstrlenA@4
      MOV   ESI,EAX
      POP   EBX
      CMP   EDI,1
      JNE   NO_ENT
;в конце - перевод строки
      MOV  BYTE PTR [EBX+ESI],13
      MOV  BYTE PTR [EBX+ESI+1],10
      MOV  BYTE PTR [EBX+ESI+2],0
      ADD  EAX,2
NO_ENT:
;вывод строки
      PUSH 0
      PUSH OFFSET LENS
      PUSH EAX
      PUSH EBX
      PUSH HANDL
      CALL WriteConsoleA@20
      RET
WRITE  ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 3.8.4:

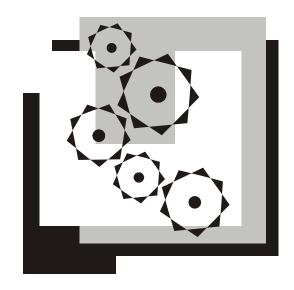
```
ML /c /coff /DMASM delserv.asm
LINK /SUBSYSTEM:CONSOLE delserv.obj
```

Данный пример показывает, как достаточно просто можно удалить сервис, даже если он находится на выполнении. Правда, это при условии, что в сервисе предусмотрена реакция на команду остановки. В принципе все достаточно очевидно. Обратите только внимание, что даже если при выполнении API-функции `ControlService` возникнет ошибка, все будет выполняться своим чередом. Дело в том, что эта команда посылает команду останова сервису. Но если сервис в базе, но не запущен, тогда возникнет ошибка с номером 1062, что означает, что сервис и так остановлен.



Часть IV

ОТЛАДКА, АНАЛИЗ КОДА ПРОГРАММ, ДРАЙВЕРЫ



Глава 4.1

Обзор инструментов для отладки и дизассемблирования

Помните программу DEBUG.EXE? Удивительно, но этот отладчик, написанный для операционной системы MS-DOS, сохранился и в Windows Vista. Жаль, что его уже не поддерживает фирма Microsoft, и сейчас он бесполезен. А как много с ним было связано. Легендарные были времена.

В этой главе мы рассмотрим отладочные и дизассемблирующие программы, кроме трех наиболее известных, о которых пойдет речь в последующих двух главах.

> ***Замечание***
>
> Материал этой и последующих глав требует от читателя дополнительных знаний структуры исполняемого модуля и в области дизассемблирования кода. Информацию по этим вопросам можно почерпнуть в книге [27].

Утилиты фирмы Microsoft

EDITBIN.EXE

Название программы многообещающе, но в действительности программу нельзя назвать редактором. Основное ее назначение — конвертировать объектные файлы в OMF-формате в объектные файлы в COFF-формате. Кроме того, данная утилита позволяет менять некоторые другие атрибуты исполняемых и объектных модулей. Если в командной строке данной программы указать имя объектного модуля, то, в случае если модуль будет в OMF-формате, он будет преобразован в COFF-формат. Рассмотрим ключи данной программы, которые можно применять как к исполняемым, так и к объектным модулям.

- `/BIND` — позволяет указать пути к динамическим библиотекам, которые используют данный исполняемый модуль. Например, `EDITBIN /BIND:PATH=c:\edit;d:\dll EDIT.EXE`.

- `/HEAP` — задает размер кучи в байтах. Например, `EDITBIN /HEAP:100000,100000` (см. опции программы LINK.EXE).
- `/LARGEADDRESSAWARE` — указывает, что приложение оперирует адресами, большими 2 Гбайт.
- `/NOLOGO` — подавляет вывод информации о программе.
- `/REBASE` — устанавливает базовый адрес модуля. По умолчанию для исполняемого модуля базовый адрес равен `400000H`, для динамической библиотеки — `10000000H`.
- `/RELEASE` — устанавливает контрольную сумму в заголовке исполняемого модуля.
- `/SECTION` — изменяет атрибуты секций исполняемого модуля. Полный формат опции: `/SECTION:`*`name`*`[=`*`newname`*`][,`*`attributes`*`][,`*`alignment`*`]`. Здесь *`attributes`* — атрибут (см. табл. 4.1.1), *`alignment`* — параметр выравнивания (см. табл. 4.1.2).

Таблица 4.1.1. *Значение атрибутов*

Атрибут	**Значение**
C	Code (секция кода)
D	Discardable (может быть выгружен из памяти)
E	Executable (исполняемый)
I	Initialized data (инициализированные данные)
K	Cached virtual memory (кэшируемые данные)
M	Link remove (удаляется при компоновке)
O	Link info (комментарий компоновщика)
P	Paged virtual memory (подвергается страничному преобразованию)
R	Read (можно читать)
S	Shared (разделяемый)
U	Uninitialized data (неинициализированные данные)
W	Write (можно изменять)

Таблица 4.1.2. *Значение опции выравнивания*

Опция	**Кратность выравнивания (в байтах)**
1	1
2	2

Таблица 4.1.2 *(окончание)*

Опция	Кратность выравнивания (в байтах)
4	4
8	8
p	16
t	32
s	64
x	Без выравнивания

- `/STACK` — изменяет значение требуемого для загружаемого модуля стека. Например:

  ```
  EDITBIN /STACK:10000,10000 EDIT.EXE
  ```

- `/SUBSYSTEM` — переопределяет подсистему, в которой работает данная программа. Например, если программа оттранслирована с опцией `/SUBSYSTEM:WINDOWS`, можно изменить установку без повторной компиляции следующей командой:

  ```
  EDITBIN /SUBSYSTEM:CONSOLE EDIT.EXE
  ```

- `/SWAPRUN` — устанавливает для исполняемого модуля атрибут "помещать модуль в SWAP-файл".
- `/VERSION` — устанавливает версию для исполняемого модуля.
- `/WS` (`/WS:AGGRESSIVE`) — устанавливает атрибут `AGGRESSIVE`, который используется операционной системой.

Утилита весьма полезна для быстрого изменения атрибутов исполняемых и объектных модулей.

DUMPBIN.EXE

Программа DUMPBIN.EXE входит в состав пакета Visual Studio .NET и используется для исследования загружаемых и объектных модулей COFF-формата, выводя информацию в текущую консоль. Разумеется, консольный вывод всегда можно перенаправить в текстовый файл, получив, таким образом, возможность подробно изучить дизассемблированный текст. Несмотря на свою консольную природу, данная программа работает довольно толково и вполне годится для анализа небольших программ.

Ключи программы:

- `/ALL` — выводить всю доступную информацию о модуле, кроме ассемблерного кода;

- `/ARCH` — выводить содержимое секции `.arch` заголовка модуля;
- `/ARCHIVEMEMBERS` — выводить минимальную информацию об элементах объектной библиотеки;
- `/DEPENDENTS` — выводить имена динамических библиотек, откуда модулем импортируются функции;
- `/DIRECTIVES` — выводить содержимое секции `.drectve`, создаваемой компилятором (только для объектных модулей);
- `/DISASM` — дизассемблировать содержимое секций модуля с использованием символьной (отладочной) информации, если она присутствует;
- `/EXPORTS` — выводить экспортируемые модулем имена;
- `/FPO` — выдавать на консоль информацию о FPO-оптимизации (frame pointer optimization, оптимизация указателя стека);
- `/HEADER` — выдавать на консоль заголовки модуля и всех его секций. В случае объектной библиотеки выдает заголовки составляющих ее объектных модулей;
- `/IMPORTS` — выводить имена, импортируемые данным модулем;
- `/LINENUMBERS` — выдавать на консоль номера строк объектного модуля, если таковые имеются;
- `/LOADCONFIG` — выводить структуру `IMAGE_LOAD_CONFIG_DIRECTORY`, которая используется загрузчиком и которая определена в файле WINNT.H;
- `/LINKERMEMBER[:{1|2}]` — выводить все имена в объектной библиотеке, определяемые как `public`;
  - `/LINKERMEMBER:1` — в порядке следования объектных модулей в библиотеке;
  - `/LINKERMEMBER:2` — вначале выдает смещение и индекс объектных модулей, а затем список имен в алфавитном порядке для каждого модуля;
  - `/LINKERMEMBER` — сочетание ключей `1` и `2`;
- `/OUT` — определяет, что вывод осуществляется не в консоль, а в файл (например, `/OUT:ED.TXT`). Конечно, перенаправить вывод в файл можно, просто используя знак `>`;
- `/PDATA` — выводить содержимое таблиц исключения (для RISC-процессоров);
- `/RAWDATA` — выдает дамп каждой секции файла. Разновидности данного ключа: `/RAWDATA:BYTE`, `/RAWDATA:SHORTS`, `/RAWDATA:LONGS`, `/RAWDATA:NONE`, `/RAWDATA:,`*`number`*. Здесь *`number`* определяет ширину строк;

- `/RELOCATIONS` — выводить все перемещения в таблице перемещений;
- `/SECTION:`*`section`* — определять конкретную анализируемую секцию;
- `/SUMMARY` — выдавать минимальную информацию о секциях;
- `/SYMBOLS` — выдавать таблицу символов COFF-файла (объектного файла).

Пример использования:

```
dumpbin  /disasm  prog.exe > prog.txt
```

В текстовый файл prog.txt будет выведен дизассемблированный код программы.

Особенностью программы DUMPBIN.EXE является то, что она дизассемблирует только секции с известными ей именами. Если вы поместите исполняемый код в секцию с произвольным (непредопределенным) именем, то программа не будет выводить дизассемблированный код, хотя дамп и выведет.

В качестве интересного примера рассмотрим исследование с помощью программы DUMPBIN.EXE *таблицы перемещений* (relocation table) динамической библиотеки. Для примера я взял очень простую динамическую библиотеку, написанную на ассемблере.

> ***Замечание***
>
> Информация, содержащаяся в таблице перемещений, может понадобиться загрузчику Windows, если по каким-либо причинам он будет загружать модуль по адресу, отличному от указанного в заголовке исполняемого модуля. Таблица содержит относительные адреса тех ячеек памяти, содержащие, в свою очередь, используемые в программе адреса, значения которых, возможно, потребуется изменить при загрузке (см. [27]).

Пусть название библиотеки prog.dll. Выполним команду:

```
dumpbin /disasm prog.dll
```

В листинге 4.1.1 представлены строки, являющиеся дизассемблированным текстом исполняемого кода модуля. Для некоторых строк я дописал также свой комментарий.

Листинг 4.1.1. Пример дизассемблирования динамической библиотеки при помощи утилиты DUMPBIN.EXE

```
10001000: B8 01 00 00 00      mov    eax,1     ;начало процедуры входа
10001005: C2 0C 00            ret    0Ch
10001008: 55                  push   ebp       ;начало экспортируемой
                                               ;функции
10001009: 8B EC               mov    ebp,esp
1000100B: 83 7D 08 01         cmp    dword ptr [ebp+8],1
1000100F: 75 13               jne    10001024
```

```
10001011: 6A 00             push   0
10001013: 68 26 30 00 10    push   10003026h
10001018: 68 3E 30 00 10    push   1000303Eh
1000101D: 6A 00             push   0
1000101F: E8 04 00 00 00    call   10001028 ;вызов функции API
10001024: 5D                pop    ebp
10001025: C2 04 00          ret    4
10001028: FF 25 00 20 00 10 jmp    dword ptr ds:[10002000h]
```

А теперь выведем таблицу перемещения и выясним, в каких командах будут подправляться адреса при загрузке данной динамической библиотеки. Для этого выполним команду:

```
dumpbin /relocations prog.dll
```

Вот результат выполнения команды:

```
BASE RELOCATIONS #4
       1000 RVA,       10 SizeOfBlock
       14   HIGHLOW           10003026
       19   HIGHLOW           1000303E
       2A   HIGHLOW           10002000
```

Интересен в первую очередь левый столбец, содержащий смещение операнда, который должен быть учтен при загрузке динамической библиотеки в память. Например, смещение 14 означает, очевидно, адрес `10001014`, т. е. мы попадаем на команду `push 10003026h`. Операнд этой команды, таким образом, представляет адрес, который должен быть откорректирован, если динамическая библиотека будет загружаться по базовому адресу, отличному от `10000000h`.

Дизассемблер W32Dasm

Данному дизассемблеру будет посвящена *глава 4.3*. Эта программа, обладающая, как и IDA Pro, возможностями отладки, по-видимому, больше не поддерживается разработчиками. Во всяком случае, версия 10, которую мы и будем рассматривать, создавалась уже, судя по всему, не авторами проекта. В Интернете вы также можете встретить тоже весьма хорошую версию 8.98.

Отладчик OllyDbg

Пакет OllyDbg представляет собой 32-битный отладчик уровня ассемблера. Он сочетает в себе как возможности отладчика, так и довольно мощного дизассемблера. Данная программа не только позволяет просматривать дизассемблированный код и выполнять отладочные действия, но и дает возможность править исполняемый код, что делает его бесценным инструментом

для *исследователей исполняемого кода*. Мы подробно остановимся на возможностях этого инструмента в *главе 4.2*.

Замечание

В литературе уже давно идет дискуссия о том, как называть людей, занимающиеся дизассемблированием и исследованием исполняемого кода. Термин "хакер" явно не подходит по двум причинам: а) в силу отрицательной окраски этого термина, б) по причине слишком расплывчивости этого понятия — хакером называют и талантливого программиста, и преступника, который проникает в чужие тайны просто подбором паролей. Термин "кодокопатель" предложенный одним из авторов слишком вульгарен и может использоваться лишь в компьютерном сленге. На мой взгляд, термин "исследователь исполняемого кода" совершенно точно обозначает то, чем занимаются эти люди и к тому же звучит достойно и уважительно.

Другие инструменты

DUMPPE.EXE

Данная программа рассматривалась нами в *главе 1.1*. Она во многом похожа на предыдущую программу DUMPBIN.EXE, но более удобна, хотя и обладает несколько меньшими возможностями.

Hiew.exe

Эта программа широко известна в среде программистов, занимающихся исследованием и исправлением исполняемого кода. Название программы происходит от фразы "Hacker's view". Основная задача, которую выполняет данная программа, — просматривать и редактировать загружаемые модули. Причем просмотр и редактирование допускается в трех вариантах: двоичный, текстовый и ассемблерный. Хороших дизассемблирующих программ создано довольно много, а вот программ, подобных этой, можно по пальцам перечесть.

Интерфейс программы весьма напоминает интерфейс редакторов такой программы, как FAR.EXE (рис. 4.1.1). Все команды выполняются при помощи функциональных клавиш (в том числе в сочетании с клавишами <Alt> или <Ctrl>). Например, нажимая клавишу <F4>, вы получаете возможность выбрать способ представления двоичного файла: текстовый, ассемблерный или двоичный. Нажимая клавишу <F3> (при условии, если вы находитесь в двоичном или ассемблерном просмотре), вы получаете возможность редактировать файл. Если же, находясь в ассемблерном просмотре, вы после <F3>

нажмете еще и <F2>, то сможете редактировать машинную команду в символьном виде. Мы не будем далее останавливаться на командах данной программы, поскольку они просты, очевидны и могут быть получены просто по нажатию клавиши <F1>, а перейдем сразу к простому примеру использования данной программы, хотя пример, скорее, относится к материалу *главы 4.6*. Чтобы слишком не загромождать рассмотрение, возьмем простую консольную программу.

```
Hiew: dexem.exe
   dexem.exe      FRO    NE 00000272 a16 --------        25099 Hiew 6.81 (c)SEN
 0000025F: C05E5F3F                   rcr        b,[bp][5F],03F ;"?"
 00000263: E55D                       in         ax,05D
 00000265: C3                         retn
 00000266: 90                         nop
 00000267: B806BC                     mov        ax,0BC06 ;"╝"
 0000026A: 014740                     add        [bx][40],ax
 0000026D: 1C25                       sbb        al,025 ;"%"
 0000026F: 4A                         dec        dx
 00000270: 5E                         pop        si
 00000271: 5A                         pop        dx
 00000272: AE                         scasb
 00000273: 16                         push       ss
 00000274: 132D                       adc        bp,[di]
 00000276: 6E                         outsb
 00000277: 98                         cbw
 00000278: 97                         xchg       di,ax
 00000279: 5E                         pop        si
 0000027A: 025F04                     add        bl,[bx][04]
 0000027D: EB18                       jmps       000000297  -----▯ (1)
 0000027F: 43                         inc        bx
 00000280: 42                         inc        dx
 00000281: B737                       mov        bh,037 ;"7"
 00000283: 80FF3C                     cmp        bh,03C ;"<"
1 16/32  2GetBlk 3Replac 4ReRead 5Base  6NextRf 7NextSr 8      9FilArg10SavSta
```

Рис. 4.1.1. Интерфейс программы Hiew.exe

В листинге 4.1.2 представлена простая консольная программа, выводящая на экран текстовую строку.

Листинг 4.1.2. Консольная программа

```
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE     equ -11
INVALID_HANDLE_VALUE  equ -1
;прототипы внешних процедур
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
```

```
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
BUF DB  "Строка для вывода",0
      LENS  DWORD ? ;количество выведенных символов
      HANDL DWORD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить HANDLE вывода
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      CMP  EAX,INVALID_HANDLE_VALUE
      JNE _EX
      MOV  HANDL,EAX
;вывод строки
      PUSH 0
      PUSH OFFSET LENS
      PUSH 17
      PUSH OFFSET BUF
      PUSH HANDL
      CALL WriteConsoleA@20
_EX:
      PUSH 0
      CALL ExitProcess@4
_TEXT ENDS
END START
```

Программа в листинге 4.1.2 проста, корректна и транслируется обычным для консольных приложений способом. Представьте теперь, что при отладке вы случайно изменили одну команду: вместо `JE` поставили `JNE`. В результате после трансляции программа перестала работать. Можно исправить ее, не прибегая к ассемблерному тексту? Конечно. Для этого вначале ее следует дизассемблировать, найти ошибку, а потом воспользоваться программой Hiew.exe. Вообще говоря, можно ограничиться только программой Hiew, т. к. она вполне корректно дизассемблирует. Однако мы нарочно проведем исправление в два этапа.

Дизассемблируем модуль при помощи программы DUMPBIN.EXE. Вот дизассемблированный текст программы (листинг 4.1.3).

Листинг 4.1.3. Дизассемблированный код программы из листинга 4.1.2

```
Dump of file cons1.exe
File Type: EXECUTABLE IMAGE

00401000: 6AF5              push    0F5h
00401002: E82B000000        call    00401032
00401007: 83F8FF            cmp     eax,0FFh
0040100A: 751E              jne     0040102A
0040100C: A316304000        mov     [00403016],eax
00401011: 6A00              push    0
00401013: 6812304000        push    403012h
00401018: 6A11              push    11h
0040101A: 6800304000        push    403000h
0040101F: FF3516304000      push    dword ptr ds:[00403016h]
00401025: E80E000000        call    00401038
0040102A: 6A00              push    0
0040102C: E80D000000        call    0040103E
00401031: CC                int     3
00401032: FF2508204000      jmp     dword ptr ds:[00402008h]
00401038: FF2500204000      jmp     dword ptr ds:[00402000h]
0040103E: FF2504204000      jmp     dword ptr ds:[00402004h]
```

По дизассемблированному коду легко обнаружить ошибку. Кстати, команду `cmp eax,0FFh` надо, естественно, понимать как `cmp eax,0FFFFFFFFh`. Запомним нужный код `83F8FF`. Запускаем программу Hiew.exe, нажимаем клавишу <F7> и ищем нужное сочетание. Далее нажимаем клавишу <F3>, затем клавишу <F2> и после заменяем команду `JNE` на `JE`. Клавиша <F9> фиксирует изменение. В результате мы исправили программу без ее повторной трансляции.

DEWIN.EXE

Программа работает в командном режиме, но по сравнению, например, с DUMPBIN.EXE обладает рядом достоинств. Главное из этих достоинств — это распознавание языков высокого уровня. Кроме того, вы сами можете писать скрипт-процедуры на предлагаемом макроязыке.

IDA Pro

Программа IDA Pro в настоящее время является одним из мощнейших дизассемблеров для Windows. На момент написания данной книги существуют

уже версии 4.8 и 4.9[1] этого продукта. В книге рассматривается версия 4.7 продукта, ничем принципиально не отличающаяся от названных (см. сайты **http://www.idapro.ru** и **http://www.idapro.com**). Замечу, кстати, что IDA Pro является также и отладчиком, но поскольку функции дизассемблирования основные для нее, мы и далее будем говорить об этой программе как о дизассемблере.

IDA Pro — один из самых мощных дизассемблеров[2]. Возможности настолько велики, что многие программисты считают его всемогущим. Работая над текстом дизассемблируемой программы, вы можете называть своими именами метки и процедуры, давать свои комментарии так, что дизассемблированный текст становится, в конце концов, ясным и понятным. Исправленный текст программы сохраняется в специальную базу и при последующем запуске, естественно, восстанавливается. Интерфейс дизассемблера IDA Pro показан на рис. 4.1.2. Мы дизассемблировали одну из наших прежних программ. Обратите, кстати, внимание на ссылку `offset WNDPROC`. Название `WNDPROC` дано уже нами в процессе анализа кода программы. Но рассмотрим все по порядку.

Если вы загрузите в IDA Pro некоторый исполняемый модуль, то в каталоге, откуда произошла загрузка, обнаружите два файла с расширениями id0 и id1. Это вспомогательные файлы виртуальной памяти, которые нужны IDA Pro для хранения используемых им данных. При выгрузке загруженного модуля (**File | Close**) оба файла исчезают. В файл с расширением id1 и именем исследуемого модуля загружается образ этого модуля. Этот образ вполне идентичен образу, загруженному в 32-битную плоскую память операционной системой Windows. Таким образом, достигается полная идентичность исследуемого модуля с модулем, исполняемым операционной системой, что, несомненно, сближает IDA Pro с отладчиками. Для каждого адреса в файле хранится 32-битная характеристика: восьмибитовая ячейка, соответствующая данному адресу, и 24-битный атрибут, определяющий различные свойства данной ячейки (а именно относится ли данная ячейка к инструкции или к данным (и какой тип данных), а также есть ли другие объекты в строке: комментарии, перекрестные ссылки, метки).

Механизмы работы с виртуальной памятью IDA Pro аналогичны механизмам, которые используются операционной системой Windows. При обращении к конкретной ячейке загружается в оперативную память (в буфер) вся страница, где эта ячейка расположена. Если же изменить ячейку памяти, то про-

---

[1] Когда эта книга уже была закончена, я получил Ida Pro версии 5.0, которая значительно отличается от предыдущих версий, но рассказать об этом я смогу только в своих новых книгах.

[2] Подробнее о дизассемблере IDA Pro см. книгу [27] автора.

исходит перезапись всей страницы виртуальной памяти. Часть страниц IDA Pro держит в оперативной памяти. Модифицированные страницы периодически сбрасываются дизассемблером на диск. В случае, когда требуется загрузить страницу, а буфер страниц полон, IDA Pro ищет среди загруженных страниц модифицированную раньше всех, сбрасывает ее на диск и загружает на ее место требуемую страницу.

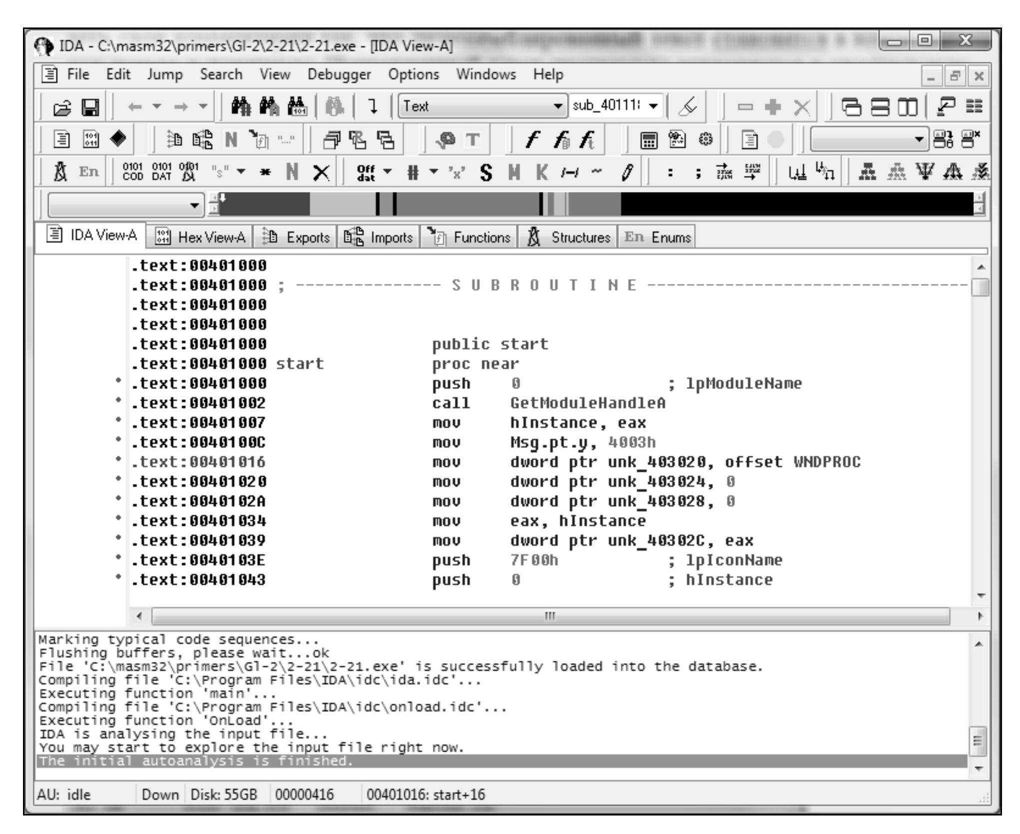
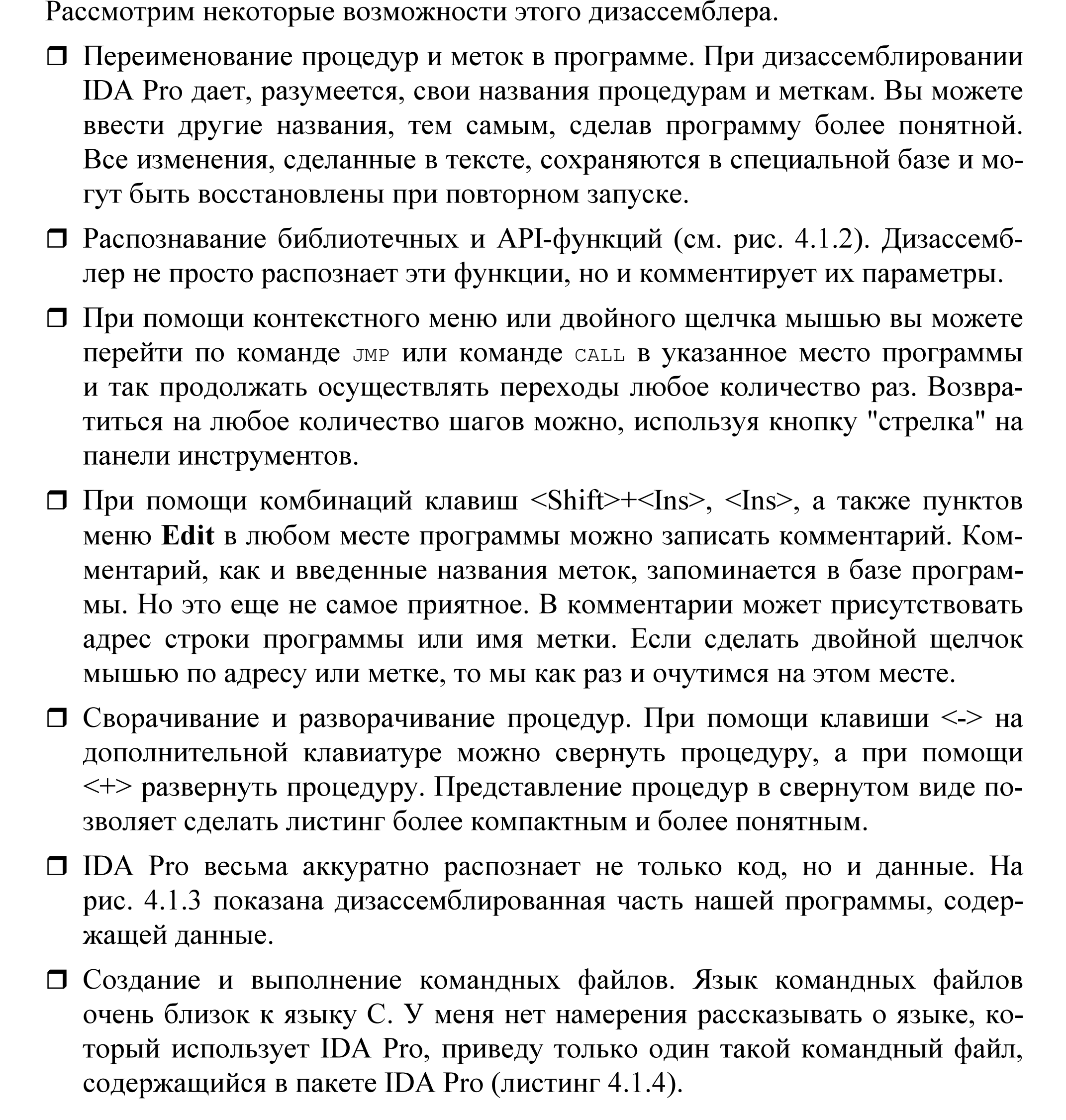


Рис. 4.1.2. Пример дизассемблирования программы с помощью самого мощного дизассемблера IDA Pro (под Windows)

Кроме хранения образа загружаемого модуля IDA Pro требуется память для хранения вспомогательной информации: имен меток, имен функций и комментариев. Для этого используется файл с расширением id0. Эту память в документации называют *memory for b-tree*[3].

[3] Balanced tree — дерево, у которого разность расстояний от корня до любых двух листьев не превышает фиксированное значение.



Рассмотрим некоторые возможности этого дизассемблера.

- Переименование процедур и меток в программе. При дизассемблировании IDA Pro дает, разумеется, свои названия процедурам и меткам. Вы можете ввести другие названия, тем самым, сделав программу более понятной. Все изменения, сделанные в тексте, сохраняются в специальной базе и могут быть восстановлены при повторном запуске.
- Распознавание библиотечных и API-функций (см. рис. 4.1.2). Дизассемблер не просто распознает эти функции, но и комментирует их параметры.
- При помощи контекстного меню или двойного щелчка мышью вы можете перейти по команде `JMP` или команде `CALL` в указанное место программы и так продолжать осуществлять переходы любое количество раз. Возвратиться на любое количество шагов можно, используя кнопку "стрелка" на панели инструментов.
- При помощи комбинаций клавиш <Shift>+<Ins>, <Ins>, а также пунктов меню **Edit** в любом месте программы можно записать комментарий. Комментарий, как и введенные названия меток, запоминается в базе программы. Но это еще не самое приятное. В комментарии может присутствовать адрес строки программы или имя метки. Если сделать двойной щелчок мышью по адресу или метке, то мы как раз и очутимся на этом месте.
- Сворачивание и разворачивание процедур. При помощи клавиши <-> на дополнительной клавиатуре можно свернуть процедуру, а при помощи <+> развернуть процедуру. Представление процедур в свернутом виде позволяет сделать листинг более компактным и более понятным.
- IDA Pro весьма аккуратно распознает не только код, но и данные. На рис. 4.1.3 показана дизассемблированная часть нашей программы, содержащей данные.
- Создание и выполнение командных файлов. Язык командных файлов очень близок к языку C. У меня нет намерения рассказывать о языке, который использует IDA Pro, приведу только один такой командный файл, содержащийся в пакете IDA Pro (листинг 4.1.4).

Листинг 4.1.4. Пример командного файла IDA Pro

```
//
// This example shows how to get list of functions.
//
#include <idc.idc>
static main() {
auto ea,x;
```

```
for (ea=NextFunction(0); ea != BADADDR; ea=NextFunction(ea) ) {
  Message("Function at %08lX:%s",ea,GetFunctionName(ea));
  x = GetFunctionFlags(ea);
  if ( x & FUNC_NORET ) Message(" Noret");
  if ( x & FUNC_FAR ) Message(" Far");
  Message("\n");
}
ea = ChooseFunction("Please choose a function");
Message("The user chose function at %08lX\n",ea);
}
```

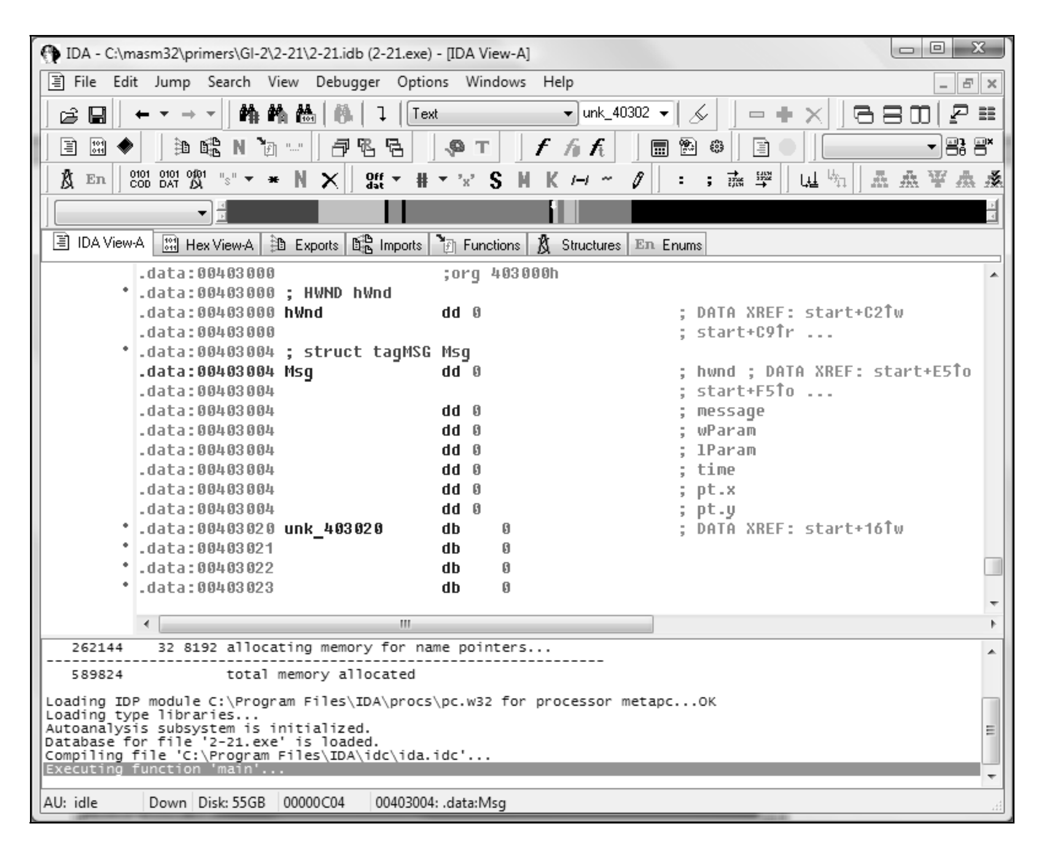
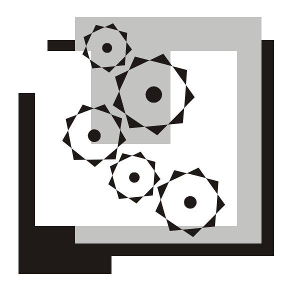


Рис. 4.1.3. Часть программы, содержащей данные, дизассемблированная при помощи IDA Pro

Прокомментирую программу из листинга 4.1.4. Как легко догадаться, организация цикла и условные конструкции имеют в точности тот же синтаксис, что и в языке С. Главное здесь — понять смысл используемых библиотечных функций. Легко видеть, что функция `Message` просто выводит

строку в окно сообщений, которое находится под основным окном. Функция `ChooseFunction` вызывает окно, которое вызывается также из меню: **Jump to Function** (функция `GetFunctionFlags` возвращает информацию об указанной функции). Наконец, функция `NextFunction` осуществляет переход на следующую функцию. Она возвращает адрес функции. Аргументом же ее является адрес функции, от которой выполняется переход на следующую функцию. Оставляю вам изучение командного языка, поддерживаемого IDA Pro, который представлен в помощи программы. Теоретически можно написать любую сколь угодно сложную программу по анализу дизассемблированного кода.

- Программа IDA Pro осуществляет дизассемблирование модулей самых различных форматов: OBJ, EXE, DLL, VXD, ZIP, NLM и др.
- Функциональность IDA Pro может быть значительно усилена посредством подключаемых модулей — plugins. Подключаемые модули пишутся на языке C++ и имеют структуру PE-модулей. Подключение модулей осуществляется через горячие клавиши или через пункты меню **Edit | Plugins**. Подключаемые модули хранятся в специальном каталоге Plugins, где находится и файл конфигурации, в котором указаны эти модули.
- Еще одна приятная особенность дизассемблера IDA Pro — он создает ассемблерный файл, с которым затем можно работать уже в текстовом режиме.



Глава 4.2

Отладчик OllyDbg

Это отличный отладчик. Например, он умеет определять параметры процедуры, циклы, выделять константы, массивы и строки, что никогда не было отличительным признаком такого рода инструментов. Отладчик поддерживает все процессоры семейства 80*x*86 и знает множество числовых форматов. Вы можете загружать в отладчик исполняемый модуль или подключаться к уже работающему процессу. В общем, возможностей — море, и мы будем о некоторых из них говорить.

Начало работы с отладчиком

Окна отладчика

Начнем рассмотрение отладчика OllyDbg с изучения главного окна этой программы (рис. 4.2.1). Кроме естественного горизонтального меню и панели кнопок, в главном окне расположены по умолчанию четыре информационных окна: окно дизассемблера (левое верхнее), окно данных (левое нижнее), окно регистров (правое верхнее), окно стека (правое нижнее). Кроме указанных окон в процессе работы можно использовать и другие окна. Перечень всех информационных окон представлен в пункте меню **View**. С частью окон вы познакомитесь в процессе изучения данного раздела, о других вы можете узнать самостоятельно, если, конечно, будете использовать данный инструмент, что я вам настоятельно рекомендую.

Обратимся теперь к окнам, которые мы видим на рис. 4.2.1. Это наиболее важные окна, без которых никак не обойтись в процессе отладки.

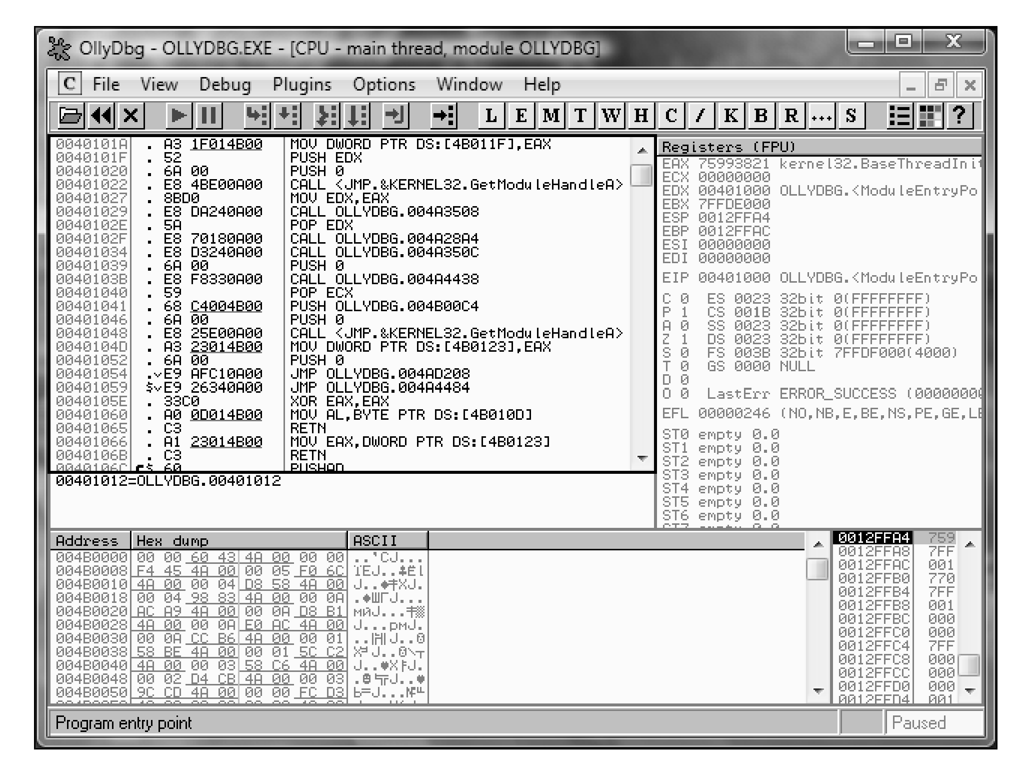


Рис. 4.2.1. Отладчик OllyDbg с загруженной в него программой

Окно дизассемблера

Окно состоит из четырех колонок.

- Колонка адреса команды (**Address**). В данной колонке показан виртуальный адрес команды, который она получает при загрузке модуля в память. Двойной щелчок мышью в данной колонке переводит все адреса в смещения относительно текущего адреса (`$`, `$-2`, `$+4` и т. п.).
- Колонка кода команды (**Hex dump**). При этом выделяются собственно код и значение операнда. Кроме этого, в колонке имеются различные значки, которые помогают разобраться в логике программы: указывают на команду, на которую есть переходы (>), на команду, осуществляющую переход (ˆ — вверх, ˇ — вниз), и т. п. В этой же колонке отмечаются циклы, которые удалось распознать программе. Двойной щелчок по этой колонке приводит к тому, что в первой колонке адрес будет подсвечен красным. Это означает, что мы установили точку останова на данную команду (адрес).

- Колонка команды (**Disassembly**). В этой колонке представлено ассемблерное обозначение команды. Двойной щелчок по колонке приводит к тому, что появляется окно редактирования ассемблерной команды. Вы можете исправить команду, и далее в отладке будет участвовать исправленная вами команда. Более того, вы можете записать исправленный текст программы в исполняемый модуль. Здорово, не правда ли?
- Колонка комментария (**Comment**). Здесь программа помещает дополнительную информацию о команде. В частности указываются имена API-функций, библиотечных функций и т. д. Сделав двойной щелчок по этой колонке, мы получим возможность добавлять свой комментарий к каждой строке ассемблерного кода.

Окно данных

Окно имеет по умолчанию три колонки: колонка адреса (**Address**), колонка, содержащая шестнадцатеричное значение ячейки (**Hex dump**), колонка текстовой интерпретации содержимого ячеек (**ASCII**, **Unicode** и т. п.). Можно менять смысл второй и третьей колонок. Например, можно интерпретировать содержимое ячеек в кодировке Unicode.

Окно регистров

Окно регистров может содержать три возможных набора: стандартные регистры и регистры сопроцессора, стандартные регистры и регистры MMX, стандартные регистры и регистры в технологии 3DNow. Двойной щелчок в этом окне позволяет редактировать содержимое соответствующего регистра.

Окно стека

Окно стека представляет содержимое стека. Первая колонка (**Address**) содержит адрес ячейки в стеке, вторая колонка (**Value**) — содержимое ячейки, третья колонка (**Comment**) — возможный комментарий к содержимому (см. рис. 4.2.1).

Еще об окнах

Приступая к работе с отладчиком, имейте в виду следующее.

- Щелкнув правой кнопкой мыши по любому из окон, вы получите контекстное меню. Оно индивидуально для каждого из четырех окон. Советую подробно изучить эти меню. Часть информации вы можете получить в процессе нашего изложения.
- Окна (их содержимое) не являются независимыми. Посмотрите на регистры. Щелкнув правой кнопкой мыши по одному из рабочих регистров,

можно всегда перевести его содержимое, как адрес в области данных (**Follow in dump**) или в области стека (**Follow in stack**).

Отладочное выполнение

Отладка — это анализ поведения программы путем исполнения ее в различных режимах. Вот о различных режимах выполнения программы в отладчике OllyDbg мы сейчас и поговорим.

Итак, исполняемый код загружен в отладчик. В окне дизассемблера мы видим ассемблерный код. Какие же основные способы выполнения программы можно использовать?

- Пошаговое выполнение с обходом процедур (step over). При нажатии клавиши <F8> выполняется текущая ассемблерная команда. Выполняя одну команду за другой, мы можем в трех остальных окнах следить за тем, как меняется содержимое регистров, секции данных и секции стека. Особенностью данной команды является то, что если очередной командой будет команда вызова процедуры (`CALL`), то автоматически будут выполняться все команды процедуры (все команды процедуры выполнятся как одна инструкция).
- Пошаговое выполнение с заходом в процедуру (step into). Выполнение осуществляется по нажатию клавиши <F7>. Основным отличием от предыдущего способа является то, что при встрече с командой `CALL` далее пошагово будут выполняться инструкции процедуры.
- Оба способа пошагового выполнения (step over и step into) можно автоматизировать, если использовать так называемую *анимацию* (animation), соответственно, нажимая комбинации клавиш <Ctrl>+<F8> или <Ctrl>+<F7>. При нажатии этих комбинаций клавиш команды "step over" и "step into" будут выполняться в автоматическом режиме одна за другой с небольшой задержкой. После каждой инструкции окна отладчика будут обновляться, так что можно отслеживать динамику изменений. В любой момент можно приостановить выполнение, нажав клавишу <Esc>. Выполнение приостанавливается также на точках останова и в случае, если исполняемая программа генерирует исключение.
- Еще один способ пошагового выполнения программы — это *трассировка* (trace). Она напоминает анимацию, но при этом на каждом шаге не обновляются окна отладчика. Два способа трассировки, соответствующие "step over" и "step into", выполняются с помощью комбинаций клавиш <Ctrl>+<F12> и <Ctrl>+<F11>. Остановить трассировку можно теми же способами, что и анимацию. После каждой команды информация о ее вы-

полнении заносится в специальный трассировочный буфер, который можно просмотреть с помощью пункта меню **View | Run trace**. При желании содержимое буфера можно сбросить в текстовый файл. Можно определить условия, по которым будет происходить остановка трассировки (**Set trace condition**) — через комбинацию клавиш <Ctrl>+<T> (точка останова). При этом можно задать:

  - диапазон адресов, в котором будет произведен останов;
  - условные выражения, например, `EAX>100000`, при выполнении которых трассировка будет остановлена;
  - номер команды или набор команд, по которым будет произведен останов.

- Можно заставить отладчик выполнить код, пока не встретится возврат из процедуры (**Execute till return**). Другими словами будет выполнен весь код текущей процедуры и осуществлен возврат из нее. Для этого предназначена комбинация клавиш <Ctrl>+<F9>.
- Наконец, если в процессе трассировки вы оказались глубоко в системном коде, можно дать команду выхода из него (**Execute till user code**) — нажать комбинацию клавиш <Alt>+<F9>.

Точки останова

Точки останова (точки прерывания, контрольные точки) — это очень мощное средство отладки приложения. Они позволяют разобраться в логике выполнения программы, давая мгновенные снимки регистров, стека и данных в определенные моменты выполнения.

Обычные точки останова

Обычные точки останова (ordinary breakpoints) ставятся на конкретную команду. Для этого в окне дизассемблера используется клавиша <F2> или двойной щелчок мыши во второй колонке окна кода (**Hex dump**). В результате адрес команды в первой колонке (**Address**) окрашивается по умолчанию в красный цвет. Этот вид точек останова, в первую очередь, помогает найти корреляцию между наблюдаемым нами ходом выполнения программы (появление окон, сообщений и т. п.) и конкретными участками программного кода. Кроме этого, в точке останова можно проверить состояние регистров, переменных, состояние стека. Вторичное нажатие клавиши <F2> в точке останова или двойной щелчок мыши удаляет точку останова. Имейте в виду, что остановка осуществляется перед выполнением "помеченной" команды.

Условные точки останова

Условные точки останова (conditional breakpoints) устанавливаются по нажатию комбинации клавиш <Shift>+<F2>. При этом появляется окно с комбинированным списком, куда можно занести точку останова. В поле комбинированного списка задается условие, при выполнении которого должна быть произведена остановка на данной команде. Отладчик поддерживает достаточно сложные выражения, содержащие условия. Приведу несколько примеров:

- `EAX==1` — остановка на отмеченной команде (перед ее выполнением) будет осуществлена, если содержимое регистра `EAX` будет равно `1`;
- `EAX=0 AND ECX>10` — остановка на отмеченной команде будет осуществлена, если содержимое регистра `EAX` будет равно `0`, а содержимое регистра `ECX` будет больше `10`;
- `[STRING 427010]=="Error"` — в данном случае выполнение программы приостановится, если по адресу `427010H` будет располагаться строка `"Error"`. Можно написать и так: `EAX=="Error"`, и тогда содержимое `EAX` будет трактоваться как указатель на строку;
- `[427070]=1231` — данное условие определяет остановку, если содержимое ячейки памяти `427070H` равно `1231H`;
- `[[427070]]=1231` — здесь используется косвенная адресация. Предполагается, что ячейка с адресом `427070H` содержит адрес другой ячейки, содержимое которой и будет сравниваться с числом `1231H`.

Условные точки останова с записью в журнал

Данный вид точек останова (conditional logging breakpoint) является расширением условных точек останова. Устанавливается по нажатию комбинации клавиш <Shift>+<F4>. Каждый раз, когда данная точка останова срабатывает, делается запись в журнале. Посмотреть содержимое журнала можно, нажав комбинацию клавиш <Alt>+<L> или выбрав из меню **View | Log**. Можно установить запись, которая станет появляться в журнале, а также указать выражение, значение которого будет записываться в журнал. Наконец, можно установить счетчик, который будет показывать, сколько раз должна быть произведена запись в журнал и нужно ли прерывать работу программы каждый раз, когда выполняются условия останова.

Точка останова на сообщения Windows

Поскольку сообщения приходят в функцию окна (точнее класса окна), то для установки точки останова на сообщение необходимо наличие окон, другими словами, оконное приложение должно быть запущено. Итак, для простоты

я загрузил в отладчик простое приложение всего с одним окном и запустил его при помощи комбинации клавиш <Ctrl>+<F8>. Через секунду окно приложения активизировалось. Кстати, обратили внимание, какая часть программы непрерывно выполняется? Правильно, цикл обработки сообщений. Чтобы выйти на функцию окна, нужно вызвать список созданных приложением окон. Это делается при помощи пункта меню **View | Windows**. Результат команды мы видим на рис. 4.2.2.

Windows

Handle	Title	Parent	WinProc	ID	Style	ExtStyle	Thread
001F0246	πхъёЄ т кьэх	Topmost			14CF0000	00000100	Main
└00050332	Default IME	001F0246			8C000000		Main
└0006030E	MSCTFIME UI	00050332			8C000000		Main

Рис. 4.2.2. Окно со списком окон, созданных приложением

Из рис. 4.2.2 можно узнать дескриптор окна, его название, идентификатор и, главное, адрес процедуры класса (столбец **ClsProc**). Последняя информация дает нам возможность обратиться непосредственно к функции окна и установить там обычную или условную точку останова. Однако при работе с оконными функциями эффективнее использовать точку останова на сообщение.

Итак, щелкнем по окну, изображенному на рис. 4.2.2, и выберем из контекстного меню пункт **Message breakpoint on ClassProc**. В появившемся окне можно установить параметры точки останова, а именно:

- из выпадающего списка выбрать сообщение. Замечу при этом, что можно выбрать:
  - не само сообщение, а событие, которое может знаменоваться несколькими сообщениями, например, создание и уничтожение окна, событие от клавиатуры и т. п.;
  - сообщения, определяемые пользователем;
- определить перечень окон, которые будут отслеживаться, на предмет поступления данного сообщения: данное окно, все окна с данным заголовком, все окна;
- определить счетчик — сколько раз будет срабатывать точка останова;

- будет или нет останавливаться выполнение программы;
- будет или нет производиться запись в журнал.

Потренируйтесь теперь сами с установкой описанных выше точек останова и проследите также за содержимым окна стека — это весьма поучительное занятие.

Точка останова на функции импорта

Список всех импортируемых имен в отлаживаемом модуле можно получить с помощью нажатия комбинации клавиш <Ctrl>+<N>. Далее щелкнув правой кнопкой мыши по окну, можно установить:

- точку останова на вызов импортируемой функции (команда **Toggle breakpoint on import**);
- условную точку останова на вызов импортируемой функции (команда **Conditional breakpoint on import**);
- условную точку останова на импорт с записью в журнал (команда **Conditional log breakpoint on import**);
- точки останова на все ссылки на данное имя (команда **Set breakpoint on every reference**);
- точки останова с записью в журнал на все ссылки на данное имя (команда **Set log breakpoint on every reference**)

или удалить все точки останова (команда **Remove all breakpoints**).

Точка останова на область памяти

Отладчик OllyDbg позволяет установить одну *точку останова на область памяти*. Выбираем окно дизассемблера или окно данных (dump). Далее используем контекстное меню и выбираем пункт **Breakpoint | Memory on access** (на доступ к памяти) или **Breakpoint | Memory on write** (на запись в память). После этого точка останова готова к использованию. Как вы понимаете, первый тип точки останова возможен и для данных, и для кода, второй — только для кода. Удалить точку останова на область памяти можно опять же из контекстного меню: **Breakpoint | Remove memory breakpoint**.

Точка останова в окне *Memory*

Окно **Memory** отображает блоки памяти, которые были зарезервированы для отлаживаемой программы или самой отлаживаемой программой. Вот в этом окне также можно установить одну точку останова. Для этого опять исполь-

зуется контекстное меню, появляющееся посредством щелчка правой кнопкой мыши и выбором пункта **Set memory breakpoint on access** (установить точку останова на доступ к памяти) или **Set memory breakpoint on write** (установить точку останова на запись в память). Удалить точку останова можно из того контекстного меню командой **Remove memory breakpoint**.

Аппаратные точки останова

Обычные точки останова используют стандартный вектор прерывания `INT 3`. Добавление таких точек останова может существенно замедлить выполнение отлаживаемой программы. Но, как известно, у микропроцессора Intel Pentium имеются четыре отладочных регистра `DR0`—`DR3` *(см. приложение 2)*. Эти регистры могут содержать четыре контрольные точки — виртуальные адреса текущей программы. Как только адрес, который использует команда, оказывается равным адресу в одном из указанных регистров, так генерируется исключение, перехватываемое отладчиком. *Аппаратные точки останова* не замедляют выполнение отлаживаемой программы, но, как видно из сказанного ранее, их может быть всего 4. Установить аппаратную точку останова можно из окна дизассемблера с помощью пункта **Breakpoint | Hardware on execution** контекстного меню либо в окне данных с помощью пунктов **Breakpoint | Hardware on access** или **Breakpoint | Hardware on access**. Удалить аппаратные точки останова можно с помощью того контекстного меню: **Breakpoint | Remove hardware breakpoints**.

Другие возможности

Окно наблюдения

В отладчике OllyDbg имеется окно для наблюдения за выражениями. С выражениями мы уже сталкивались, когда рассматривали условные точки останова. Вы можете использовать сколь угодно сложные выражения, в которых участвуют ячейки памяти и регистры. Окно наблюдения вызывается командой меню **View | Watches**. Щелкнув в появившемся окне правой кнопкой мыши и выбрав пункт **Add Watches** (добавить наблюдение), вы можете определить выражение, за которым отладчик будет наблюдать, т. е. выводить значение этого выражения. На рис. 4.2.3 представлено окно наблюдения, содержащее список из четырех выражений, значения которых отслеживаются при каждом выполнении команды процессора и отображаются в окне.

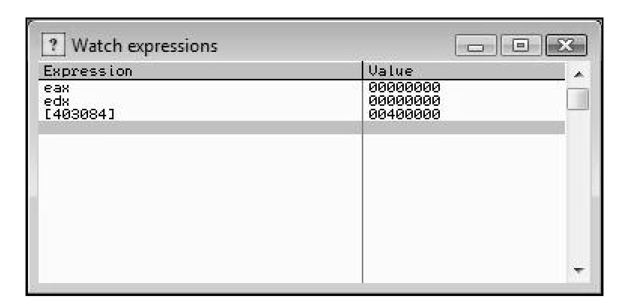


Рис. 4.2.3. Окно наблюдения за выражениями

Поиск информации

Отладчик OllyDbg позволяет эффективно искать различного рода информацию. Рассмотрим некоторые возможности.

По команде от нажатия комбинации клавиш <Ctrl>+<B> появляется окно поиска, где вы можете определить строку, которая будет разыскиваться в загруженном в отладчик модуле. Строку для поиска можно вводить в виде последовательности символов, байтов, символов в кодировке Unicode.

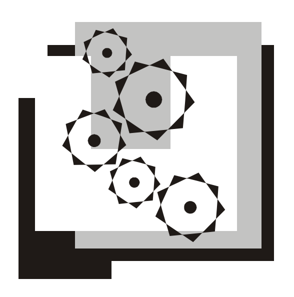
Для поиска команд используются комбинации клавиш <Ctrl>+<F> для одиночной команды и <Ctrl>+<S> для последовательности команд.

Нажатие комбинации клавиш <Ctrl>+<L> повторяет последний сделанный поиск.

Исправление исполняемого модуля

Отладчик OllyDbg обладает великолепной возможностью записи исправления в исполняемый модуль. Вы можете не только переписать с исправлениями отлаживаемый модуль, но и создать новый исполняемый модуль. Делается это очень просто. Для этого щелкаем правой кнопкой мыши в окне дизассемблера и выбираем пункт **Copy to execution | Selection**. В результате весь дизассемблированный модуль вместе с исправленными командами будет скопирован в новое окно. После этого опять щелкаем по этому окну правой кнопкой мыши и выбираем пункт **Save file**. Далее вы можете выбрать, под каким именем будет сохранен (создан) исполняемый модуль. Это действительно очень удобно: во-первых, вы можете создавать произвольное количество версий исправленного кода, во-вторых, проверка правильности исправления осуществляется, не выходя из отладчика.

На этом я закончу рассмотрение отладчика OllyDbg, хотя остается еще огромное количество интересных вопросов, связанных с использованием этой замечательной программы. Увы, все в этом мире заканчивается, и объем книги требует перехода к следующим вопросам.



Глава 4.3

Описание работы с дизассемблером W32Dasm и отладчиком SoftICE

В данной главе я попытаюсь рассказать о двух моих любимых средствах анализа программ.

Отладчик W32Dasm

Программа W32Dasm (Windows Disassembler) представляет собой симбиоз довольно мощного дизассемблера и отладчика. Версия 8.93 программы, наиболее распространенная в настоящее время, может работать не только с PE-модулями, но и DOS-, NE-, LE-модулями. Я намерен довольно полно описать работу с этой программой.

Начало работы

Внешний вид программы представлен на рис. 4.3.1. Меню дополняется панелью инструментов, элементы которой активизируются в зависимости от ситуации.

Как уже было сказано, программа является дизассемблером и отладчиком в одном лице. Это отражено также в двух пунктах меню: **Disassembler** и **Debug**. Соответственно, имеются отдельные настройки для дизассемблера и отладчика. Для дизассемблера существуют всего три опции, касающиеся анализа перекрестных ссылок в условных переходах, безусловных переходах и вызовах процедур. По умолчанию все три опции установлены. Отмена этих опций нежелательна, т. к. снижает информативность дизассемблированного текста. В принципе, отмена указанных опций может понадобиться при дизассемблировании очень большой программы, чтобы несколько ускорить процесс анализа кода программы.

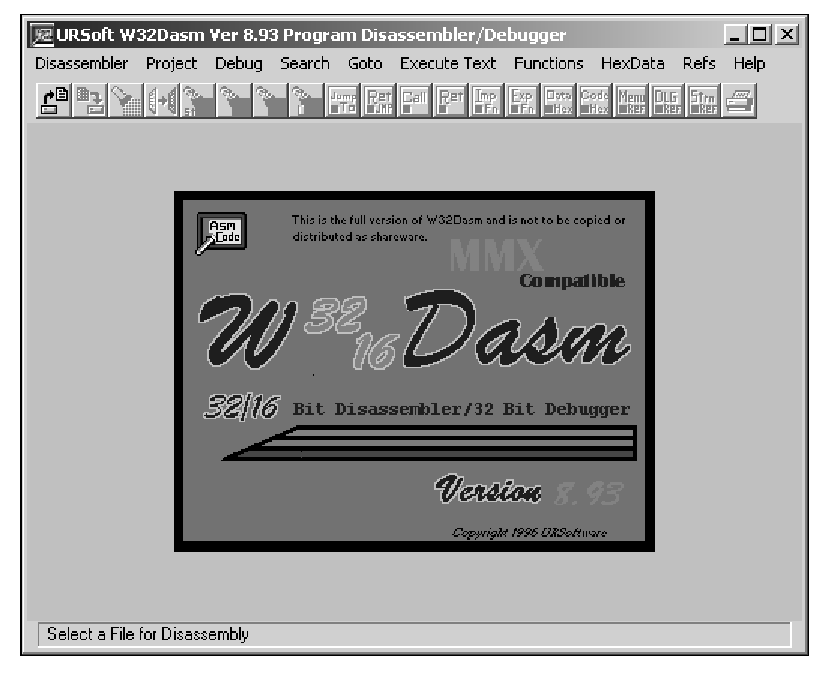


Рис. 4.3.1. Интерфейс программы W32Dasm

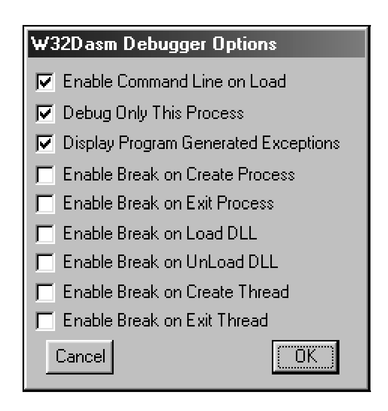


Рис. 4.3.2. Опции отладчика

Опций отладчика несколько больше, но они все очевидны. Окно установки опций отладчика изображено на рис. 4.3.2, все они касаются особенностей загрузки процессов, потоков и динамических библиотек.

Для начало работы с исполняемым модулем достаточно выбрать нужный файл в меню **Disassembler | Open File...**. После этого программа производит

анализ модуля и выдает дизассемблированный текст, а также весьма полную информацию о секциях модуля[1]. W32Dasm весьма корректно распознает API-функции и комментирует их (рис. 4.3.3).

```
* Possible Reference to String Resource ID=00001: "!>>1I5=85"
                                  |
:00401013 6A01                    push 00000001
:00401015 FF3518304000            push dword ptr [00403018]

* Reference To: USER32.LoadStringA, Ord:01A8h
                                  |
:0040101B E8AE000000              Call 004010CE
```

Рис. 4.3.3. Фрагмент дизассемблированного текста

После работы с модулем можно создать проект работы при помощи пункта **Disassembler | Save Disassembler...**. По умолчанию проект сохраняется в подкаталог wpjfiles, который расположен в рабочем каталоге W32Dasm, и состоит из двух файлов: с расширением alf — дизассемблированный текст, с расширением wpj — собственно сам проект (W32Dasm Project). При повторном запуске можно открывать уже не модуль, а проект с помощью пункта **Project | Open...**.

Передвижение по дизассемблированному тексту

При передвижении по тексту текущая строка подсвечивается другим цветом, при этом особо выделяются переходы и вызовы процедур. Передвижение облегчается также с помощью пункта меню **Goto**:

- **Goto Code Start** — переход на начало листинга;
- **Goto Program Entry Point** — переход на точку входа программы, наиболее важный пункт меню;
- **Goto Page** — переход на страницу с заданным номером, по умолчанию число строк на странице составляет пятьдесят;
- **Goto Code Location** — переход по заданному адресу, в случае отсутствия адреса учитывается диапазон и близость к другим адресам.

Другим способом передвижения по дизассемблированному тексту является пункт **Search** — поиск. Здесь нет никаких отличий от подобных команд других программ.

[1] Хотя W32Dasm работает с модулями разного типа, мы рассматриваем только модули формата PE.

В случае, если текущая строка находится в команде перехода или вызова процедуры, с помощью кнопок, расположенных на панели инструментов, можно перейти по соответствующему адресу. Такое передвижение можно продолжать, пока вы не обнаружите требуемый фрагмент программы. Но самое приятное здесь то, что можно передвигаться и в обратном направлении. При этом нужные кнопки на панели инструментов автоматически подсвечиваются.

Кроме того, те адреса, куда производится переход, содержат список адресов, откуда производятся переходы. Подсветив строку, где расположен адрес, и дважды щелкнув правой кнопкой мыши по этому адресу, мы перейдем к соответствующей строке.

Отображение данных

Есть несколько вариантов работы с данными.

Во-первых, имеется пункт меню **HexData | ex Display of Data...**, где можно просмотреть содержимое сегментов данных в шестнадцатеричном и строковом варианте. Кроме того, сам код программы также можно просматривать в шестнадцатеричном виде. Для этого используется пункт **HexData | Hex Display of Code...**.

Во-вторых, имеется пункт меню **Refs | String Data References**. Это весьма мощное и полезное средство. При выборе этого пункта появляется список строк, на которые имеются ссылки в тексте программы. Во всяком случае, это то, что сумел определить дизассемблер при анализе программы.

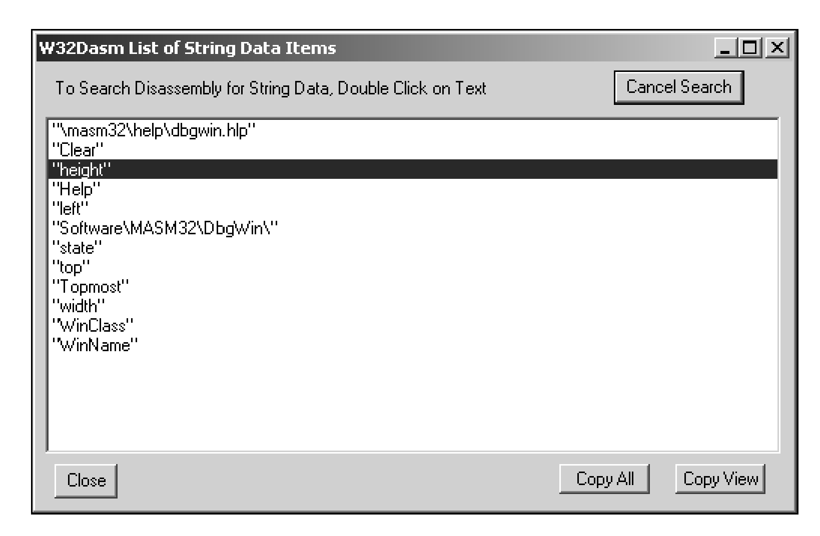


Рис. 4.3.4. Окно ссылок на строки

Выбрав нужную строку, можно двойным щелчком перенестись в соответствующее место программы. Если ссылок на данную строку несколько, то, продолжая делать двойные щелчки, мы будем переходить во все нужные места программы. На рис. 4.3.4 изображено окно ссылок на строковые типы данных.

Как видно из рисунка, можно скопировать в буфер выбранную строку или все строки.

Вывод импортированных и экспортированных функций

Список импортированных функций и модулей находится в начале дизассемблированного текста (рис. 4.3.5). Кроме того, список импортированных функций можно получить из меню **Functions | Imports**. Выбрав нужную функцию в списке, двойным щелчком можно получить все места программы, где вызывается эта функция.

Экспортированные функции также можно получить в соответствующем окне, выбрав пункт **Functions | Exports**.

```
++++++++++++++++++ IMPORTED FUNCTIONS ++++++++++++++++++
Number of Imported Modules =    7 (decimal)

   Import Module 001: ADVAPI32.dll
   Import Module 002: KERNEL32.dll
   Import Module 003: MPR.dll
   Import Module 004: COMCTL32.dll
   Import Module 005: GDI32.dll
   Import Module 006: SHELL32.dll
   Import Module 007: USER32.dll

++++++++++++++++++ IMPORT MODULE DETAILS +++++++++++++++

   Import Module 001: ADVAPI32.dll

 Addr:000D9660 hint(0000) Name: RegCloseKey
 Addr:000D966E hint(0000) Name: RegOpenKeyExA
 Addr:000D967E hint(0000) Name: RegQueryValueExA
 Addr:000D9692 hint(0000) Name: RegSetValueExA

   Import Module 002: KERNEL32.dll
```

Рис. 4.3.5. Фрагмент списка импортированных модулей и функций

Отображение ресурсов

В начале дизассемблированного текста также описаны и ресурсы, точнее, два основных ресурса — меню и диалоговое окно. Со списком этих ресурсов можно работать и в специальных окнах, получаемых с помощью пунктов меню программы **Refs | Menu References** и **Refs | Dialog References**. Строковые

ресурсы можно увидеть в уже упомянутом окне просмотра перечня строковых ссылок (см. рис. 4.3.4). Остальные ресурсы данной версии программы, к сожалению, не выделяются.

Операции с текстом

Строки дизассемблированного текста могут быть выделены и скопированы в буфер либо напечатаны. Выделение строки осуществляется щелчком левой кнопки мыши, когда курсор мыши расположен в крайнем левом положении. Для выделения группы строк дополнительно используется клавиша <Shift>. Выделенный фрагмент копируется специальной кнопкой, которая "загорается", когда фрагмент существует либо отправляется на печатающее устройство.

Загрузка программ для отладки

Загрузить модуль для отладки можно двумя способами. С помощью пункта **Debug | Load Process** загружается для отладки уже дизассемблированный модуль. Пункт же **Debug | Attach to an Active Process** позволяет "подсоединяться" и отлаживать процесс, находящийся в памяти. После загрузки отладчика на экране появляются два окна. Первое окно — информационное (рис. 4.3.6), в документации оно называется "нижним левым окном отладчика". Второе окно — управляющее (рис. 4.3.7), называемое в документации "нижним правым окном отладчика".

Информационное окно содержит несколько окон-списков: содержимое регистров микропроцессора, значение флагов микропроцессора, точки останова, содержимое сегментных регистров, история трассировок, история событий, базовые адреса, два дисплея данных. Далее я объясню также значения кнопок этого окна.

Обратимся теперь к управляющему окну. Кнопка **Run F9** запускает загруженную в отладчик программу, кнопка **Pause** приостанавливает работу программы, кнопка **Terminate** останавливает выполнение программы и выгружает ее из отладчика. Кнопки **Step Over F8** и **Step Into F7** используются для пошагового исполнения программы. Первая кнопка, выполняя инструкции, "перескакивает" код процедур и цепочечные команды с повторением, вторая кнопка выполняет все инструкции последовательно. Кроме того, имеются кнопки **AutoStep Over F6** и **AutoStep IntoF5** для автоматического пошагового выполнения программы. В случае API-функций даже использование кнопки **Step Into F7** не приведет к пошаговому выполнению кода функции в силу того, что код функции не доступен для пользовательских программ. Очень удобно, что при пошаговом выполнении происходит передвижение не только в окне отладчика, но и в окне дизассемблера.

```
Eip:00401000 is in Module: dbgwin.exe

Regs  Copy
eax=00000000
ebx=7ffdf000
ecx=00010101
EDX=FFFFFFFF
esi=00000000
edi=00000000
ebp=0012fff0
esp=0012ffc4

O D I T S Z A P C
0 0 1 1 0 1 0 1 0

#Processes: 001
#Threads: 001

Running
Paused
SS Thru
SS Into
Terminated

Source For Data Disp 1   Copy
eip  [esp-00000014] - 00000000  ....
eax  [esp-00000010] - 77e87900  .y.w
ebx  [esp-0000000C] - fffffffe  ....
ecx  [esp-00000008] - 00000009  ....
edx  [esp-00000004] - 0012fff8  ....
esi  [esp+00000000] - 77e87903  .y.w
edi  [esp+00000004] - 00000000  ....
ebp  [esp+00000008] - 00000000  ....
esp  [esp+0000000C] - 7ffdf000  ....
UA1  [esp+00000010] - 7ffde000  ....
UA2  [esp+00000014] - 0012ffc8
On Off Mode-> DWord Word Byte Code

Segment Reg
cs=001b
ds=0023
ss=0023
es=0023
fs=0038
gs=0000

BPts Copy Clear
None Set
AA
DA

User Addr 1
00400000
User Addr 2
00401000

Source For Data Disp 2   Copy
Disp 1
UA1
UA2
Oper
Address: 77E87903 is in Module: KERNEL32.dll
 char[000]:""
 DWORD:030a47e9, WORD:47E9, BYTE:e9
 CODE: jmp 00431A62

Active DLLs Copy
ADVAPI32.dll
COMCTL32.dll
dbgwin.exe

Proc Create Brk
Proc Exit Brk
Thrd Create Brk
Thrd Exit Brk
DLL Load Brk
DLL UnLoad Brk

Tr0001 Process @ Program Entry Point, eip:0040100(  Copy
Ev0008 Loading DLL RPCRT4.DLL @ addr:77d40000  Copy
Modify Data    Goto Current Eip
```

Рис. 4.3.6. Первое информационное окно отладчика

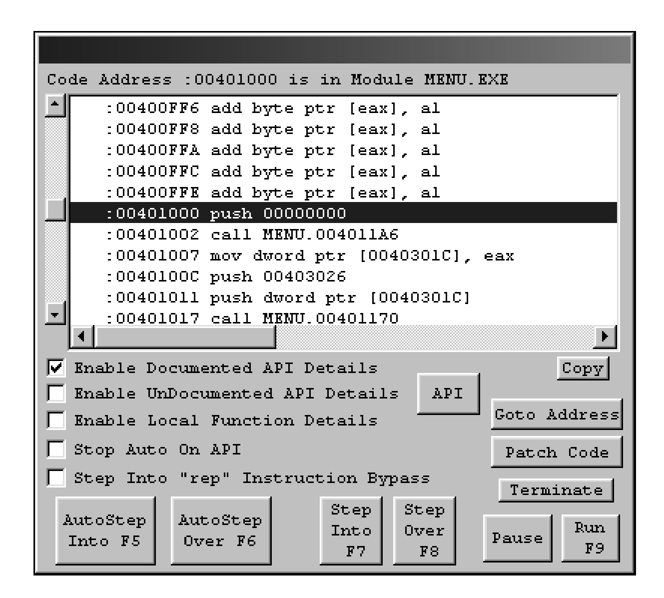


Рис. 4.3.7. Второе управляющее окно отладчика

Отмечу, что если вы подсоединяетесь к процессу, расположенному в памяти, то при выходе из отладчика процесс также будет выгружен из памяти, что может привести к неправильной работе операционной системы.

Работа с динамическими библиотеками

Для отладки динамической библиотеки можно поступить следующим образом. Загрузить в отладчик программу, которая обращается к динамической библиотеке. Затем обратиться к списку используемых динамических библиотек. Возможно, для работы с данной динамической библиотекой вам понадобится запустить программу и выполнить какую-либо ее функцию. Дважды щелкнув по нужной библиотеке, вы получите дизассемблированный код данной библиотеки в окне дизассемблера и возможность работать с кодом библиотеки.

Точки останова

В дизассемблированном тексте можно установить точки останова. Для этого следует перейти к нужной строке и воспользоваться клавишей <F2> или использовать левую кнопку мыши при нажатой клавише <Ctrl>. Установка точки останова в окне дизассемблера тут же отражается в информационном окне и в управляющем окне — у отмеченной команды появляется префикс `BP*`. Удалить точку останова можно тем же способом, что и при установке. Точку останова можно сделать также неактивной. Для этого нужно обратиться к информационному окну и списку точек останова. Выбрав нужный адрес, щелкните по нему правой кнопкой мыши. При этом "звездочка" у данной точки останова исчезнет, а строка в окне дизассемблера из желтой станет зеленой.

Быстрый переход к точке останова можно произвести, выбрав ее из списка (информационное окно) и сделав двойной щелчок мышью. Наконец можно установить точки останова на определенные события, такие как загрузка и выгрузка динамической библиотеки, создание и удаление потока и т. д. Все это делается при помощи установки соответствующего флага в информационном окне.

Модификация кода, данных и регистров

Отладчик позволяет модифицировать загруженный в него код (рис. 4.3.8). Сделать это можно, обратившись к кнопке **Patch Code** в управляющем окне (см. рис. 4.3.7). Важно отметить, что модификации подвергается только код, загруженный в отладчик, а не дизассемблированный текст. Найдя нужное

место в отлаживаемом коде и модифицировав его, вы можете тут же проверить результат модификации, запустив программу. Если модификация оказалась правильной, можно приступать уже к модификации самого модуля.

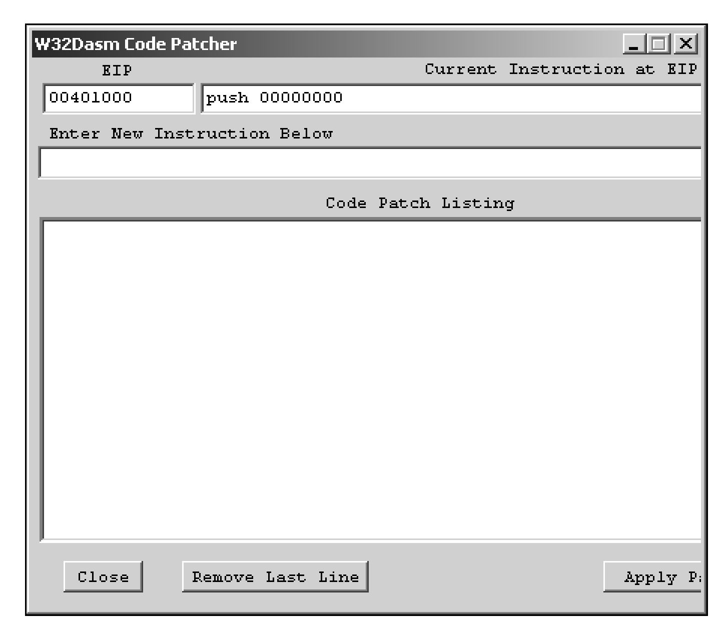


Рис. 4.3.8. Окно модификации отлаживаемого кода

Для модификации регистров и ячеек памяти исполняемого процесса, существует специальная кнопка **Modify Data** на панели информационного окна (см. рис. 4.3.6). Окно изображено на рис. 4.3.9. Оно несколько загромождено элементами, но, присмотревшись, вы поймете, что все элементы на своем месте. В верхней части окна расположены текущие значения основных флагов микропроцессора, которые вы можете изменить. Для того чтобы модифицировать содержимое регистра или ячейку памяти, следует вначале установить модифицирующую величину — **Enter Value ->**. Далее следует выбрать нужный регистр и нажать кнопку слева от него. Чтобы установить старое значение, следует нажать кнопку **R** справа от регистра. Чтобы изменить содержимое ячейки памяти, необходимо вначале записать адрес ячейки в поле **Mem Lock**, а затем воспользоваться кнопкой **Mem**. Другие операции, предоставляемые данным окном, также достаточно очевидны.

Отладчик позволяет выдавать дополнительную информацию о выполняемых API-функциях. Чтобы воспользоваться этим, необходимо сделать следующее. В управляющем окне установите флаги: **Enable Documented API Details**,

Stop Auto On API (см. рис. 4.3.7). Далее запустите программу на выполнение клавишей <F5>. При прохождении API-функции будет производиться остановка и на экране будет появляться окно с информацией о данной функции.

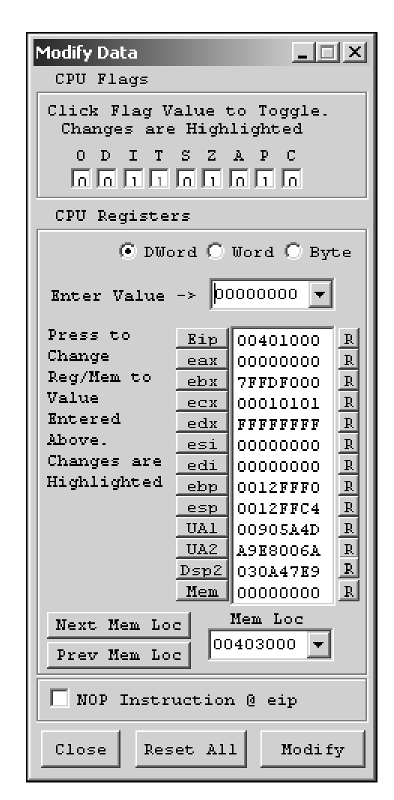


Рис. 4.3.9. Окно модификации регистров и ячеек памяти. Дополнительные возможности для работы с API

Поиск нужного места в программе

Часто требуется найти в дизассемблированном коде место, соответствующее месту исполняемой программы. Наиболее эффективно это можно сделать следующим образом. Загружаем в отладчик данный модуль. Запускаем его, доходим до нужного места и нажимаем кнопку **Terminate**. В результате подсвеченная строка в дизассемблированном коде окажется как раз в нужном месте. Следует только иметь в виду, что некоторые программы делают изменения, которые потом продолжают действовать. К таковым относятся, в частности, горячие клавиши.

К использованию программы W32Dasm мы еще вернемся в последующих главах.

Отладчик SoftICE

В настоящее время отладчик SoftICE существует для всех операционных систем Windows и даже MS-DOS. Прежде всего, заметим, что SoftICE — это отладчик уровня ядра (kernel mode debugger). С его помощью можно отлаживать любые программы, работающие в операционной системе, в том числе сервисы и драйверы, работающие в нулевом кольце защиты. По причине тесного взаимодействия отладчика с операционной системой, с его помощью можно получить много системной (я бы сказал, довольно личной) информации о функционировании операционных систем. Поэтому SoftICE просто незаменим для тех, кто изучает внутренние механизмы функционирования операционной системы Windows. В среде исследователей кода SoftICE считается лучшим[2] отладчиком.

Сам отладчик дополняется также утилитами, главная из которых — это Symbol Loader (загрузчик отладочной информации). Программа Symbol Loader (loader32.exe) загружает исполняемый модуль в память и осуществляет вызов окна отладчика SoftICE, другими словами, устанавливает точку останова на точку входа программы. При наличии в исполняемом модуле отладочной информации, распознаваемой загрузчиком, он также загружает ее в отладчик. Отладчик позволяет отлаживать исполняемый код не только на автономном компьютере, но и производить удаленную отладку (remote debugger) — с удаленного компьютера, соединенного с первым посредством COM-порта.

Установка SoftICE — это отдельная статья. Поскольку отладчик работает на уровне ядра, то разработчикам постоянно приходится дорабатывать свой продукт, дабы его можно было использовать со всеми релизами операционных систем Windows. И, тем не менее, Интернет полон статей и обсуждений, посвященных проблемам установки SoftICE и устранению различных проблем во время работы этого продукта. Чтобы не загромождать книгу, я опущу вопросы установки отладчика и отошлю заинтересованного читателя к сайту поддержки **http://www.compuware.com**, где в частности после регистрации вы можете получить руководства по SoftICE (Reference Guide). Отмечу также русскоязычный сайт **http://www.wasm.ru**, где можно найти материалы и обсуждения, касающиеся различных технических проблем, возникающих при установке SoftICE.

Моя задача, которую преследую в данной главе, — дать вводное руководство по отладке приложений при помощи SoftICE. По этой причине я настроен

---

[2] Название отладчика SoftICE — это указание на то, что отладчик в любой момент может "заморозить" (от англ. *to ice* — замораживать) систему и дать полную информацию и по системе и по всем, работающим в ней приложениям.

довольно полно описать команды SoftICE, чаще всего используемые при отладке обычных приложений, а также дать примеры отладки при наличии в исполняемом модуле отладочной информации и в отсутствии последней.

В настоящей главе все примеры и описания рассматривались мной по отношению к операционным системам Windows XP и Windows Server 2003.

Основы работы с SoftICE

В данном разделе я предоставлю читателю основные сведения, которые помогут ему начать работу с этим мощным инструментом.

Запуск и интерфейс

Главное окно SoftICE

При появлении окна SoftICE все системные функции оказываются "замороженными", поэтому для того, чтобы изобразить окно SoftICE, мне пришлось поставить рядом два компьютера и срисовать его внешний вид (рис. 4.3.10).

```
EAX=FFDFFC50     EBX=FFDFF000     ECX=0002A182     EDX=80010031     ESI=8054A6A0
EDI = 8054A900   EBP=80541F50     ESP=80541F44     EIP=806CEFAA     o d I s z A p C
CS=0008   DS=0023   SS=0010   ES=0023   FS=0030   GS=0000
----------------------------------------------------------------------------
ST0  0                                ST4  0
ST1  0                                ST5  0
ST2  0                                ST6  0
ST3  0                                ST7  0
--------WinRar!.text+2344-------------------------byte-------PROT-------(0)----
010:00403344  C8 8B D6 E8 34 8D 00 00 - FF 45 C0 BA 02 00 00 00  ....4....E......
010:00403354  66 C7 45 B4 2C 00 6A 00 - 6A 00 8B C6 E8 F7 55 00  f.E.,.j.j.....u.
010:00403364  00 8D 95 80 FB FF FF 8B - CB 8B C6 E8 6C 53 00 00  ............is..
010:00403374  8D 95 80 FB FF FF 8B CB - 8B 45 80 E8 88 9C 00 00  .........E......
-------------------------------------------------------------------PROT32----
01B:00473D0B  NOP
01B:00473D0C  PUSH        EBP
01B:00473D0D  MOV         EBP,ESP
01B:00473D0F  PUSH        EBX
01B:00473D10  MOV         EBX,[EBP+08]
01B:00473D13  CALL        ↑00401140
01B:00473D18  ADD         EAX,0000001C
FrameEBP---RetEIP---Syms---Symbol-------------------------------------------
0012FFB8  00401040  N    WinRAR!.text+00072D0C
0012FFF0  00000000  N    WinRAR!.text+0040
(PASSIVE)--KTEB(84D4A020)--TID(0648)---WinRAR!.text+00072D0B----------------
11DC2000              84ED9968  0230  LSASA
11FA0000              84DE6670  02DC  SVHOST
12D01000              84D60110  0324  SVHOST
13DF2000              84F56000  03BC  SVHOST
13FF4000              84D71000  03D8  SPOOLSV
14DD0000              84D45600  0438  DSRSVC
:
:
:■
Enter a command (H for HELP)                                         WinRar
```

Рис. 4.3.10. Главное окно SoftICE

Окно отладчика SoftICE, которое мы будем называть еще главным окном, может появляться в четырех случаях:

- по нажатию комбинации клавиш <Ctrl>+<D>. Данная команда приведет к вызову окна отладчика в любом случае, при выполнении любой программы. Вы, таким образом, можете в любой момент посмотреть состояние системы и исполняемых приложений;
- при загрузке какого-либо приложения в память с помощью программы Loader32.exe. При этом происходит прерывание процесса загрузки как раз на точке входа в исполняемый модуль, и вы можете продолжить выполнение приложения в каком-либо из режимов отладчика с самого начала;
- при выполнении условия одной из точек останова, которую вы установили ранее в окне отладчика. Отладчик покажет при этом то самое место, которое стало причиной прерывания. Большим преимуществом отладчика SoftICE является возможность работы одновременно с несколькими приложениями. Вы можете устанавливать точки останова сразу для нескольких приложений;
- кроме этого, окно SoftICE может появляться при возникновении системной ошибки или крахе системы (синий экран).

Итак, окно отладчика SoftICE представлено на рис. 4.3.10. В главном окне располагаются еще несколько окон, которые содержат различную информацию. Количество таких окон может быть различно. Вы по своему усмотрению можете сами добавлять или удалять эти окна в главное окно. На нашем рисунке представлены наиболее востребованные окна отладчика. Особо отмечу, что вы можете не только просматривать информацию в этих окнах, но и менять их содержимое, например содержимое регистров процессора. Однако делать это следует с большой осторожностью, т. к. это может привести к непредсказуемому поведению приложения и, самое главное, всей системы. Итак, обратимся к рассмотрению окон отладчика, переходя по очереди от самого верхнего окна к нижним окнам (см. рис. 4.3.10).

- **Окно регистров.** В окне перечисляются все регистры, в том числе и сегментные регистры (кроме регистров сопроцессора), и их содержимое. Представлен в окне и регистр флагов, причем каждый флаг обозначен отдельной буквой. Если флаг изменился в последней операции, то он изображается заглавной буквой и другим цветом (цвет на черно-белом рисунке изобразить невозможно).
- **Окно регистров сопроцессора.** В окне представлено содержимое всех восьми регистров сопроцессора.
- **Окно данных.** Окно предназначено для представления содержимого области памяти, как в байтовом, так и в ASCII-виде. Можно прокручивать

содержимое окна, просматривая, таким образом, произвольные области памяти.

- **Окно кода.** В окне представлен дизассемблированный код, который также может прокручиваться в окне. Если загружаемое вами приложение содержит отладочную информацию, распознаваемую SoftICE, то в этом окне вы увидите и текст программы на языке высокого уровня.
- **Окно стека.** В окне стека представлено не все содержимое стека, а только стековый кадр, связанный непосредственно с работой данного приложения.
- **Командное окно.** В окне можно набирать многочисленные команды отладчика SoftICE. В частности из рисунка мы видим, что, набрав команду `H`, можно получить помощь: список команд отладчика. Для получения информации по конкретной команде следует набрать `H` и имя команды, например, так: `H hwnd`.

При работе в главном (командном) окне для управления отладчиком можно использовать команды, которые, как я уже сказал, можно набирать в командном окне, и специальные управляющие сочетания клавиш. Кроме этого, предусмотрено использование стандартной мыши и контекстного меню.

Обратите также внимание на самое нижнее окно — это окно или панель подсказки. При наборе в командной строке какой-либо команды окно подсказки поможет вам правильно набрать эту команду и ее параметры. В частности будут перечислены все команды, которые начинаются с тех символов, которые вы уже набрали. Кроме этого, в правом нижнем углу окна отладчик всегда показывает текущий процесс. На этот важный момент всегда обращайте внимание, чтобы не перепутать приложения. Мы вернемся к этому вопросу позднее.

Кроме перечисленных выше окон можно использовать также **окно слежения** — в нем отслеживается значения переменных, которые указаны в команде `WATCH`, **окно регистров MMX**, **окно локальных переменных**.

Режимы работы отладчика

После установки отладчика SoftICE вы можете выбрать пять способов запуска:

- **Disable** — отладчик не запускается;
- **Manual** — отладчик не запускается автоматически. Для запуска следует применить команду `Net start ntice`. В каталоге, куда инсталлируется SoftICE, есть пакетный файл ntice.bat с этой командой. Данный режим наиболее безопасен, но в нем невозможно выполнять отладку драйверов устройств на этапе загрузки;

- **Automatic** — отладчик запускается автоматически. В этом режиме, однако, нельзя отлаживать драйверы режима ядра;
- режимы **System** и **Boot** — в обоих случаях отладчик запускается автоматически. Отличие режимов друг от друга заключается в порядке загрузки системных и загрузочных драйверов.

Загрузчик (Loader)

На рис. 4.3.11 представлено главное окно программы loader32.exe, предназначенной для загрузки в отладчик исполняемых модулей. Данная утилита умеет также извлекать из отлаживаемых модулей отладочную информацию, если она имеется, и передавать ее отладчику SoftICE. Загружая отлаживаемый модуль, данная утилита устанавливает точку останова на вход в программу.

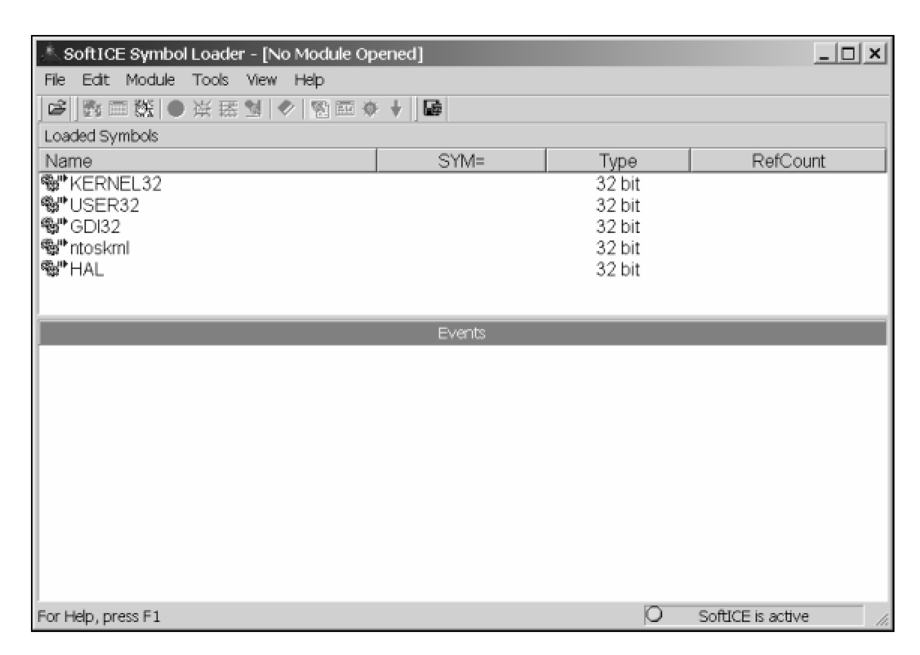


Рис. 4.3.11. Программа загрузчик loader32.exe

Загрузка исполняемого модуля

Для того чтобы загрузить в отладчик исполняемый модуль, следует:

1. Открыть его при помощи пункта меню **File | Open...**. Можно для этой цели воспользоваться кнопкой **Open** на панели инструментов.

2. Далее выбрать пункт меню **Module | Load**. Можно также воспользоваться кнопкой **Load Symbols** на панели инструментов. При этом загрузчик вначале транслирует найденную им символьную информацию в файл с расширением nms и таким же именем, как имя программы, а после загружает исходный исполняемый модуль вместе с отладочной информацией в отладчик SoftICE. В случае, если отладочная информация отсутствует, загрузчик сообщит об этом и предложит выбрать: загружать или нет исследуемый модуль в отладчик. Трансляцию отладочной информации (т. е. создание файла с расширением nms) можно осуществить и отдельной командой: при помощи пункта меню **Module | Translate** или используя кнопку **Translate** на панели инструментов.

   Список **Loaded Symbols** содержит названия загруженных модулей. Обратите внимание на столбец **SYM=**. При загрузке исполняемого модуля здесь будет отображаться объем загруженной символьной информации. Модули, не содержащие отладочной символьной информации, в список **Loaded Symbols** не заносятся.

Параметры загрузки

После того как модуль, предназначенный для исследования в отладчике SoftICE, открыт, можно установить параметры загрузки. Для этого используется пункт меню **Module | Settings...**. Окно, где можно установить параметры загрузки, изображено на рис. 4.3.12. Окно содержит четыре вкладки. Разберем их подробнее.

- Вкладка **General**.
  - В поле редактирования **Command line arguments** можно задать параметры командной строки, с которыми будет запускаться в отладчике исследуемая программа.
  - В поле редактирования **Source file search path** указываются пути поиска файлов, которые связаны с отлаживаемым модулем.
  - В поле редактирования **Default source file search path** задается основной путь для поиска файлов. Отладчик всегда вначале ищет файлы согласно полю **Source file search path** и только потом использует данное поле.
  - Когда флажок **Prompt for missing source files** установлен, то загрузчик сообщит вам, если не все файлы, необходимые для отладки исполняемого модуля, могут быть загружены. В частности, если отсутствует отладочная информация, вам будет предложено продолжить или прервать выполнение загрузки исполняемого модуля в отладчик.

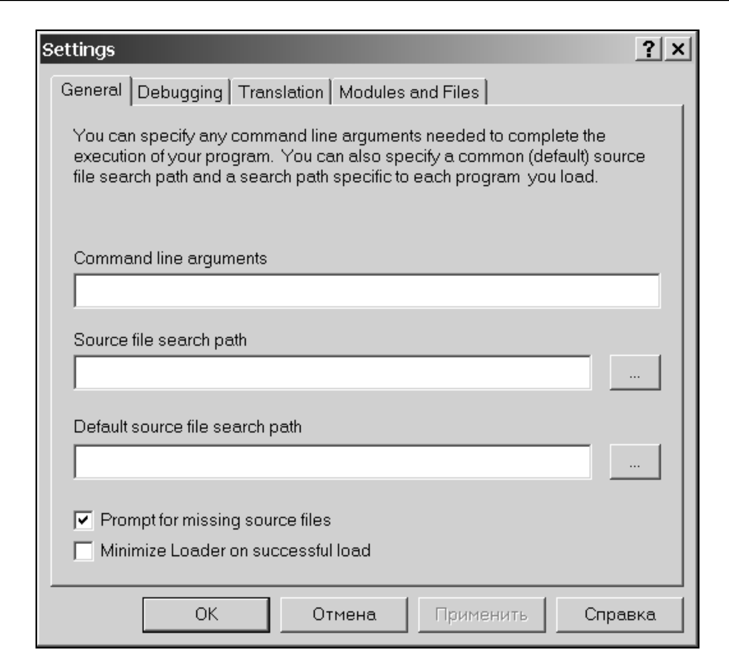


Рис. 4.3.12. Окно установки параметров загрузки отлаживаемых модулей

- Флажок **Minimize Loader on successful load** используется для минимизации загрузчика в памяти после загрузки исполняемой программы в отладчик.

- Вкладка **Debugging** позволяет менять некоторые текущие параметры отладки.
  - Переключатели **Load symbol information only** и **Load executable** позволяют загружать в отладчик только отладочную информацию либо и отладочную информацию и исполняемый модуль соответственно.
  - Флажок **Stop at WinMain, main, DllMain etc.** позволяет устанавливать точку прерывания в начало пользовательской части исполняемого модуля. В отсутствии отладочной информации точка прерывания устанавливается в начало выполнения программы.

- С помощью вкладки **Translation** задаются параметры трансляции отладочной информации исполняемого модуля.

  Переключатели:

  - **Publics only** — транслировать только внешние имена;
  - **Type information only** — транслировать только информацию о типах переменных;

- **Symbols only** — транслировать только символьные имена;
- **Symbols and source code** — транслировать всю отладочную информацию;
- **Package source with symbol table** — сохранять оттранслированную информацию в файле формата NMS.

❒ На вкладке **Modules and Files** можно перечислить все файлы и их местоположение, которые будут загружены вместе с загружаемым исполняемым модулем. Вы можете перечислить здесь все файлы, содержащие отладочную информацию. Специальным переключателем можно временно заблокировать загрузку того или иного файла.

Некоторые приемы работы с SoftICE

Начало работы. Процессы

Рассмотрим основные моменты работы с SoftICE.

Мы работаем в многозадачной операционной системе. Программа, которую вы хотите исследовать при помощи SoftICE, после загрузки станет всего лишь одним из многих процессов. Вы должны четко знать, с каким процессом вы работаете. Не перепутайте процессы, это может привести к зависанию всей системы. Отладчик показывает текущий процесс в правом нижнем углу окна подсказки.

При загрузке приложения при помощи программы loader32.exe остановка происходит на начале выполнения программы. При этом созданный процесс оказывается текущим. Так что вы можете спокойно трассировать загруженное приложение *(см. разд. "Команды трассировки" далее в этой главе)*. Однако когда вы закроете окно отладчика (клавиша <F5>) и снова вызовете его, то данный процесс уже не будет текущим. Каждый запущенный процесс имеет свое виртуальное адресное пространство (контекст процесса). Все команды, так или иначе связанные с адресами, относятся к конкретному адресному пространству. Например, команда `D DS:004080AF` выдаст содержимое памяти для конкретного виртуального адресного пространства — для текущего процесса. Для того чтобы работать с адресами конкретного процесса, следует сделать процесс текущим. Для этого используйте команду `ADDR` (описание команды *см. в разд. "Основные информационные команды" далее в этой главе*):

```
: ADDR 058
```

Здесь `058` — это идентификатор процесса (Process Identifier, PID), значение его можно узнать, если использовать команду `ADDR` без параметров.

Основным средством исследования исполняемого кода являются точки останова. Надо четко понимать, куда вы ставите точку останова, т. е. к какому процессу (потоку) относится данная точка останова. В частности, это касается точек останова на вызов функции API. Когда создаете такую точку останова, всегда при помощи условной конструкции указывайте, к какому процессу она относится. Для этого используйте функцию `PID`, которая возвращает текущий идентификатор процесса. Значение же идентификатора можно получить с помощью все той же команды `ADDR`. Пример создания условной точки останова на функцию API `CreateWindowEx`:

```
: BPX CreateWindowEx  if(PID==0x58)
```

Еще раз подчеркну, что значение идентификатора для интересующего нас процесса можно определить при помощи команды `ADDR` или `PROC`. Для того чтобы установить такую точку останова, совсем не требуется делать интересующий нас процесс текущим.

Точки останова

При исследовании конкретного исполняемого кода одной из задач всегда является поиск нужного места в программе. При отсутствии текста на языке высокого уровня, а так почти всегда и бывает, если вы только не отлаживаете свой собственный программный проект, незаменимым средством являются точки останова.

Одноразовые точки

Одноразовые точки останова функционируют только один раз. Фактически такая точка останова — это строка в окне кода, на которую указывает курсор (подсвеченная строка). Перемещается курсор при помощи команды `U`. Команда `HERE` (или нажатие клавиши <F7>) выполняют исполняемый код, начиная с текущей команды и до отмеченной таким образом строки. Имейте в виду, что команда `HERE` набирается из окна кода, куда вы должны предварительно перейти (клавиша <F6>). Можно также воспользоваться командой `G` *`address`*, и тогда код будет выполняться до адреса *`address`*.

Постоянные точки останова

Типичным примером постоянной точки останова является точка останова на конкретную команду (конкретный виртуальный адрес процесса). Для этого следует выйти в окно кода и использовать команду `BPX` без параметров. Вы можете двигаться по коду и устанавливать точки останова по нужным адресам. При этом строки, на которые устанавливаются точки прерывания, подсвечиваются. Точно такого же результата можно достигнуть, если воспользоваться клавишей <F9>. Убрать точку останова можно также повторным

указанием команды `BPX` на уже поставленной точке останова. Аналогично можно использовать и клавишу <F9>.

К данным точкам останова применима общая схема управления точками останова: `BL` — получить список точек останова с номерами, `BC` *n* — удалить точку останова с заданным номером, `BC *` — удалить все точки останова, `BE` *n* — редактирование точки останова с заданным номером. Наконец, если вы знаете адрес, куда вы хотите поставить точку останова, то можно использовать этот адрес в команде `BPX`. Например, `BPX 0008:806CEFAB`. Разумеется, повторное использование команды `BPX` с тем же адресом удаляет точку останова. Не забывайте, что точка останова на адрес команды относится к конкретному адресному пространству, т. е. конкретному процессу.

Условные точки останова

В условных точках останова указываются те условия, при наступлении которых точка должна активизироваться. Невозможно поставить две точки останова на один адрес или на одну API-функцию, но вы можете при помощи условных конструкций учесть разные варианты вызова одной и той же точки останова.

Рассмотрим типичные примеры использования условных точек останова.

Пример 1.

Точка останова на конкретный адрес срабатывает только, если содержимое регистра `EAX` принимает указанное значение.

```
: BPX  0008:806CEFAB if(EAX==406090)
```

Пример 2.

Маленькое исследование с точкой останова на вызов функции `MessageBox` (рассмотрено приложение WinRar). После запуска приложения WinRar вызовем окно SoftICE и определим идентификатор приложения, используя команду `ADDR`. Идентификатор оказался равным `0x328`. Пишем следующую команду создания условной точки останова:

```
: BPX  MessageBoxA  if(PID==0x328)
```

Проверим командой `BL`, что точка останова задана, как и положено. Заметим, что мы указали суффикс `A` — это обязательно, так SoftICE различает функции API по их истинному имени.

Выйдем из отладчика, нажав клавишу <F5>, и выполним одну из команд программы, которая должна вызвать отображение окна `MessageBox`. Тут же появится окно SoftICE. В командном окне мы обнаружим сообщение о причине появления SoftICE. В данном случае мы видим:

```
Break due to BP 00: USER32!MessageBoxA  IF(PID==0x328) (ET=2.65 seconds)
```

Обратим внимание теперь на окно кода. Там подсвечена первая строка входа в процедуру `MessageBox`:

```
USER32!MessageBoxA
001B:77D56471    CMP      DWORD  PTR [77D8C3D0],0
```

Теперь мы запросто можем исследовать стек вызова функции `MessageBoxA` и получить адрес возврата и значения параметров. Выполнив например команду `? *(ESP+4)`, получим значение дескриптора окна, которое и инициализировало вызов `MessageBox` (если вам это не понятно, обратитесь к *главе 1.2*). Значение `HWND` оказывается равным `100EC`. Просмотрев список окон приложения WinRar с помощью команды `HWND 328`, мы убеждаемся, что такое окно действительно есть и соответствует оно классу `WinRarWindow`. Кстати, здесь же в таблице мы видим и адрес функции данного окна, так что можем запросто углубиться в изучение работы этого окна. Но вернемся снова к первой строке вызова функции `MessageBox` и найдем теперь адрес возврата. На адрес возврата, разумеется, указывает регистр `ESP`, и командой `? *(ESP)` мы получаем, что он равен `43C76D`.

Впрочем, адрес возврата можно получить и другим способом: воспользоваться клавишей <F11>, и после появления окна `MessageBox` и нажатия одной из кнопок этого окна мы опять оказываемся в SoftICE на строке, следующей за вызовом функции `MessageBox`.

> ***Замечание***
>
> Вообще поиск по вызову той или иной функции API — дело достаточно тонкое. Надо хорошо знать эти функции и понимать, что один и тот же результат достигается разными способами. Скажем, вам надо узнать, где создается окно. "Нужно искать `CreateWindow`", — скажете вы. А вот и нет.
>
> Во-первых, в действительности функции `CreateWindow` нет. На самом деле, даже если вы в своей программе вызываете `CreateWindow`, то все равно используется `CreateWindowEx`.
>
> Во-вторых, искать надо не `CreateWindowEx`, а `CreateWindowExA` и `CreateWindowExW`.
>
> В-третьих, окно могло быть создано модальными диалоговыми функциями `DialogBoxIndirect`, `DialogBoxParam`, `DialogBoxIndirectParam` или не модальными функциями `CreateDialogParam`, `CreateDialogIndirect`, `CreateDialogIndirectParam`. Причем для всех функций надо учитывать суффиксы `A` и `W`.

Пример 3.

Отслеживание содержимого регистров.

```
: BPX EIP IF(EAX==0x10)
```

Прерывание по данной точке останова сработает, когда значение регистра `EAX` будет равно `0x10`, вне зависимости от того, в каком потоке будет происходить данное событие.

Прерывания на сообщения Windows

Как мы знаем, основное действо в приложениях GUI разворачивается в оконных функциях. Как реагирует функция на то или иное сообщение — важнейшая задача исследования. И вот тут незаменимую услугу может оказать точка прерывания на сообщение Windows. Вот пример установки такой точки:

```
: BMSG  100EC WM_CREATE
```

Первым параметром команды оказывается дескриптор окна, на функцию которого должно прийти сообщение. Отладчик по значению дескриптора окна определяет поток, который создал это окно, так что об этом можно не заботиться. После прихода нужного нам сообщения будет вызван SoftICE, а в окне кода будет показано начало функции окна. Интересно, что того же результата можно добиться и с помощью обычной команды `BPX`:

```
: BPX  43C76D IF((ESP->8)==WM_CREATE)
```

Первым параметром мы указали адрес первой команды функции окна. А далее мы воспользовались тем фактом, что второй параметр функции находится на расстоянии 8 байтов от вершины стека.

Поиск процедуры окна

Итак, как же можно выйти на процедуру окна? Вот несколько простых советов.

- Просмотрите список окон приложения, который выводится командой `HWND` *n*, *n* — идентификатор приложения (идентификатор приложения, как вы уже знаете, можно получить с помощью команды `ADDR`). В списке окон имеются названия, иногда по ним легко определить нужное окно и, соответственно, адрес процедуры.
- Список может оказаться небольшим, и вы легко можете протестировать все процедуры, поставив в начале процедуры (на одну из первых команд) точку останова. В случае, если при активизации окна срабатывает точка останова — следовательно, это и есть нужное нам окно.
- Проанализируйте работу окна на предмет того, какие функции API могут вызываться при работе с этим окном (так, в частности, мы поступили в примере 2 из предыдущего раздела). Поставьте точку останова на эту функцию и поработайте с этим окном. При прерывании определите, откуда вызывалась данная функция. Это и будет функция окна. Кроме этого, имейте в виду, что многие функции API первым параметром содержат дескриптор окна.

Если приложение содержит отладочную информацию

SoftICE — полноценный отладчик, т. е. он может загружать отладочную информацию и представлять ее вместе с исполняемым кодом. Таким образом,

его можно использовать при отладке собственных приложений вместо стандартного отладчика, встроенного в интегрированную среду. Рассмотрим, например, как это делается при программировании на C++ в Visual Studio .NET.

При установке опции "добавить отладочную информацию" (для этого лучше всего выбрать конфигурацию проекта "DEBUG") вместе с исполняемым модулем создается база отладочной информации. База представляет собой файл, по умолчанию имеющий то же имя, что и исполняемый модуль и имеющий расширение pdb (см. также *главу 1.4*). Информации, хранящейся в файле, достаточно, чтобы представить структуру исходной программы вместе с именами глобальных и локальных переменных и сопоставить эту структуру машинному коду.

При загрузке исполняемого модуля при помощи загрузчика loader32.exe загружается и отладочная информация и передается отладчику. По умолчанию, если для программы имеется отладочная информация, то SoftICE представляет в окне кода текст программы без ассемблерных команд. В дальнейшем при помощи команды `SRC` вы можете переключиться в смешанное представление программы (текст программы и машинный код) или же к чисто машинному коду. В первом случае пошаговое выполнение программы означает ее пооператорное выполнение, в смешанном представлении шаг — это одна машинная команда. Соответственно точки останова можно устанавливать как на операторы языка высокого уровня, так и на машинные команды. Вот несколько строк, которые получены из окна кода SoftICE для случая смешанного представления:

```
00006              a=10;
001B:00411A2E      MOV DWORD PTR [EBP-a],0000000A
00007              b=11;
001B:00411A35      MOV DWORD PTR [EBP-b],0000000B
00008              c=10;
001B:00411A3C      MOV DWORD PTR [EBP-c],0000000C
00009              printf("%d\n",max(a,b,c));
001B:00411A43      MOV EAX,[EBP-c]
001B:00411A46      PUSH EAX
...
```

Разумеется, читатель понимает, что в записи типа `[EBP-a]` величина `a` — это адрес переменной `a` в стеке, точнее, смещение относительно адреса, где находится старое значение `EBP`, т. е. просто 4.

Краткий справочник по SoftICE

Справочник содержит большую часть команд отладчика SoftICE, которых более чем достаточно для исследования исполняемого кода.

Горячие клавиши

Управление экраном

- <Ctrl>+<D> — вызов или закрытие главного окна SoftICE.
- <Ctrl>+<Alt>+<*стрелки*> — перемещение главного окна SoftICE на экране с шагом, равным размеру символа.
- <Ctrl>+<Alt>+<Home> — перемещение главного окна SoftICE в левый верхний угол экрана.
- <Ctrl>+<Alt>+<End> — перемещение главного окна SoftICE в левый нижний угол экрана.
- <Ctrl>+<Alt>+<PageUp> — перемещение главного окна SoftICE в правый верхний угол экрана.
- <Ctrl>+<Alt>+<PageDn> — перемещение главного окна SoftICE в правый нижний угол экрана.
- <Ctrl>+<L> — обновление главного окна SoftICE.
- <Ctrl>+<Alt>+<C> — размещение главного окна SoftICE в центре экрана.

Перемещение внутри главного окна

- <Alt>+<C> — переход в окно кода из командного окна и обратно.
- <Alt>+<D> — переход в окно данных из командного окна и обратно.
- <Alt>+<L> — перемещение в окно локальных переменных из командного окна и обратно.
- <Alt>+<R> — перемещение в окно регистров из командного окна и обратно.
- <Alt>+<W> — переход в окно слежения из командного окна и обратно.
- <Alt>+<S> — перемещение в окно стека из командного окна и обратно.

Переход в любое (кроме окна сопроцессора) из окон главного окна отладчика можно осуществить также щелчком левой кнопки мыши в соответствующем окне.

Перемещение содержимого окон

- <↑> — перемещение на одну строку назад.
- <↓> — перемещение на одну строку вперед.
- <←> — перемещение на один символ влево.
- <→> — перемещение на один символ вправо.
- <PageUp> — перемещение на одну страницу назад.

- <PageDn> — перемещение на одну страницу вперед.
- <Home> — перейти к первой строке кода.
- <End> — перейти к последней строке кода.

Управление командным окном

- <Enter> — завершение командной строки и выполнение набранной команды.
- SoftICE помнит 32 введенных команды. Перемещение по командам, находящимся в буфере, осуществляется клавишами <↑>, <↓>. При этом учитывается набранный уже в командной строке префикс. Например, если вы набрали букву "B", то будут появляться только команды, начинающиеся на эту букву. Если вы находитесь в окне кода, то для просмотра буфера команд следует использовать сочетания <Shift>+<↑>, <Shift>+<↓>.
- При редактировании командной строки используются следующие клавиши:
  - <Home> — перевести курсор на начало командной строки;
  - <End> — перевести курсор на конец командной строки;
  - <Insert> — переключение режимов вставки/замены;
  - <Delete> — удаление символа, располагающегося справа от курсора, со сдвигом фрагмента строки влево;
  - <Backspace> — удаление символа, располагающегося слева от курсора, со сдвигом фрагмента строки влево;
  - <←>, <→> — перемещение курсора по строке.
- Отладчик SoftICE имеет *буфер протокола окна команд*. Этот буфер содержит всю информацию, выводимую ранее в окне. Просмотреть содержимое буфера можно при помощи клавиш <PageDn> и <PageUp>.

Функциональные клавиши

- <F1> — выдает помощь (равносильна команде `H`).
- <F2> — открыть/закрыть окно регистров.
- <F3> — переключение между режимами исходного кода.
- <F4> — показать экран отлаживаемого приложения.
- <F5> — вернуться в отлаживаемую программу.
- <F6> — перевести курсор в окно кода или из него.
- <F7> — выполнить отлаживаемое приложение до команды, на которую указывает курсор.

- <F8> — выполнить текущую команду отлаживаемого приложения с заходом в функции.
- <F9> — установить точку останова на текущую команду.
- <F10> — выполнить текущую команду процессора с обходом функции.
- <F11> — перейти в вызывающую функцию программы.
- <F12> — выполнить функцию до выхода в вызывающую программу.
- <Shift>+<F3> — изменить формат вывода информации в окне данных.
- <Alt>+<F1> — открыть/закрыть окно регистров.
- <Alt>+<F2> — открыть/закрыть окно данных.
- <Alt>+<F3> — открыть/закрыть окно кода.
- <Alt>+<F4> — открыть/закрыть окно слежения.
- <Alt>+<F5> — очистить содержимое окна команд.
- <Alt>+<F11> — показать данные, расположенные по адресу, размещенному в первом двойном слове окна данных.
- <Alt>+<F11> — показать данные, расположенные по адресу, размещенному во втором двойном слове окна данных.

Замечание

Список команд отладчика, который можно получить при помощи нажатия клавиши <F1> или команды `H`, довольно обширен, но содержит не все команды. Полный список команд можно найти в фирменном руководстве SoftICE Command Reference, которое можно найти на сайте **http://www.compuware.com** и других сайтах Интернета. Я в своей книге отталкиваюсь от списка, который выдает отладчик по команде `H`. Этих команд более чем достаточно, чтобы отлаживать и исследовать прикладные программы.

Макрокоманды отладчика SoftICE

Команды, о которых мы будем говорить в данном разделе, могут объединять в макрокоманды (макросы). Существуют два вида макрокоманд, которые могут присутствовать в отладчике SoftICE. Рассмотрим вначале макрокоманды времени исполнения. Эти команды существуют только в текущей сессии отладчика. После перезапуска эти команды пропадают. Вот набор команд, с помощью которых можно управлять такими макрокомандами.

- `MACRO` *`имя_макрокоманды`* `= "команда1;команда2;..."` — создание или изменение макрокоманды. Например, следующая команда создает макрокоманду с именем `_ap`:

```
: MACRO _ap  "bc *;bpx MessageBox"
```

- MACRO *имя_макрокоманды* * — удаление макроса с заданным именем. Например,
  - : MACRO _ap * — удаляет макрос _ap из списка макросов;
  - MACRO * — удаляет из списка все макрокоманды.
- MACRO *имя_макрокоманды* — редактирование макрокоманды с данным именем.
- MACRO — вывод списка макрокоманд.

Можно определять макрокоманды с параметрами. Для этого используется символ %. После данного знака следует указать номер параметра. Номер должен лежать в диапазоне от 1 до 8. Например, команда MACRO _bpx = "bpx %1;bl" создает макрос _bpx с одним параметром. Данный макрос создает точку останова на указываемую в качестве параметра команду и выводит список точек останова. Для того чтобы вставить в определение макрокоманды знак " или знак %, следует использовать символ обратной косой черты \. Для вставки же косой черты используется последовательность \\.

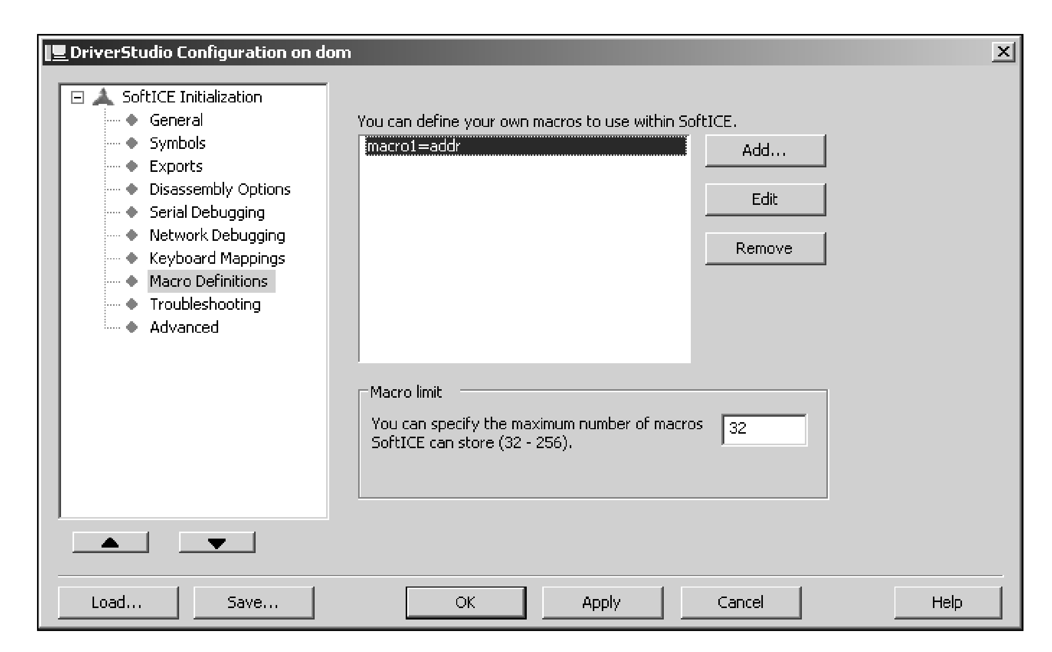


Рис. 4.3.13. Окно настройки создания постоянных макрокоманд

Для создания постоянных макрокоманд можно воспользоваться программой loader32.exe. Для этого нам понадобится пункт меню **Edit | SoftICE Initialization Settings...**. При выборе данного пункта появляется окно настроек SoftICE. В этом окне следует выбрать раздел **Macro Definitions** (рис. 4.3.13).

Дальнейшие действия довольно очевидны. Кнопки **Add...** и **Edit** используются для добавления и редактирования макросов соответственно. Кнопка **Remove** — для удаления макроса. Запомните, что все изменения, которые вы делаете в окне настроек отладчика SoftICE, вступают в действия только после перезагрузки SoftICE.

Команды управления окнами SoftICE

- `Lines` *n* — команда задает количество строк в главном окне отладчика. Значение *n* от 25 до 60.
- `Width` *m* — команда задает ширину главного окна отладчика в символах. Значение *m* в промежутке от 80 до 160.
- `Set font` *n* — команда задает размер шрифта, используемого отладчиком. *n* может принимать значения 1, 2, 3.
- `Set origin` *x y* — с помощью данной команды можно задать положение левого верхнего угла главного окна на экране.
- `Set forcepalette [on | off]` — если значение параметра равно `on`, то блокируется изменение системной палитры цветов.
- `Color [c1 c2 c3 c4 c5]|[reset]` — задает цветовую гамму главного окна отладчика. Команда `Color reset` возвращает цветовую гамму к исходному состоянию, заданному по умолчанию. Однобайтовые параметры `c1`, `c2`, `c3`, `c4`, `c5` задают цвет букв и фона соответствующего элемента главного окна отладчика. Первый полубайт задает цвет фона, второй — цвет букв.
  - `c1` — цвет основного фона и букв;
  - `c2` — цвет фона и букв для вывода изменившихся флагов (в окне регистров);
  - `c3` — цвет фона и букв для выделения текущей команды в окне кода;
  - `c4` — цвет фона и букв на панели подсказки;
  - `c5` — цвет фона и букв разделительных линий между окнами.
- Команды открытия и закрытия окон:
  - `WC` — окно кода;
  - `WD` — окно данных. Может существовать одновременно несколько окон данных. Номер окна можно указать через точку. Например, так: `wd.3`;
  - `WF` — окно сопроцессора;
  - `WL` — окно локальных переменных;
  - `WR` — окно регистров;

- ww — окно слежения;
- ws — окно стека;
- wx — окно регистров MMX.

Каждая из перечисленных команд открывает или закрывает (если окно уже есть) соответствующее окно. При этом размеры главного окна не меняются, так что появление или удаление соответствующего окна идет за счет размеров уже имеющихся окон. Вы можете также задать размер (количество строк) окна, если укажете параметр в команде, например: `wd 30` — команда задает количество строк в окне данных.

- EC — переход между окном команд и окном кода (эквивалентно использованию клавиши <F6>).
- CLS — по этой команде будет очищено командное окно. Эквивалентна комбинации клавиш <Ctrl>+<F5>.
- RS — с помощью данной команды можно временно убрать с экрана окно SoftICE. При нажатии любой клавиши происходит восстановление окна SoftICE. Команда эквивалентна нажатию клавиши <F4>.
- ALTSCR — команда предназначена для перенаправления окна SoftICE на дополнительный монитор. Формат команды:

```
ALTSCR [mono|vga|off]
```

Назначение параметров:

- mono — монохромный монитор;
- vga — монитор, который поддерживает VGA-режимы;
- off — выключить альтернативный монитор (по умолчанию).

- FLASH — команда предназначена для восстановления экрана после команд T и P. Формат команды: `FLASH on` — включить режим восстановления, `FLACH off` — выключить режим восстановления. При выполнении команды без параметров происходит вывод на экран в текущем режиме.

Получение и изменения информации в окнах

- R — команда получения и изменения информации, хранимой в регистрах. Формат команды:

```
R [-d|reg_name|reg_name [=] value]
```

Параметры:

- вариант `R -d` — просто выдает список регистров и их содержимое в окно команд;
- вариант `R reg_name` — переводит курсор в окно регистров в содержимое указанного в команде регистра. При этом вы можете редактировать со-

держимое, закрепляя исправление клавишей <Enter>. Разумеется, в окно регистров можно перейти другим способом, например при помощи мыши, и точно так же редактировать содержимое регистров;

- вариант `R` *`reg_name = value`* (знак `=` можно опустить) заносит в указанный регистр значение *`value`*.

❒ `U` — вывод в командное окно дизассемблированного листинга. Формат команды:

```
U [address [L length]]
```

Параметры:

- *`address`* — адрес, с какого предполагается вывод листинга. Можно указывать регистр, откуда этот адрес будет взят;
- *`length`* — количество выводимых в листинге байтов (длина).

При указании длины листинг выводится в командное окно. Если длину не указывать, но задать адрес, то это приводит просто к тому, что листинг в окне кода будет начинаться с указанного адреса. Команда без параметров приводит к прокрутке содержимого в окне кода, начиная с текущего адреса (на котором закончился предыдущий листинг). Если окно кода отсутствует, то весь вывод информации осуществляется в командное окно.

❒ `D` — команда вывода области памяти (дампа памяти). Формат команды:

```
D[size] [address [L length]]
```

Параметры:

- *`size`* может принимать значения: `B` — побайтовый вывод, `W` — вывод словами, `D` — вывод двойными словами, `S` — вывод короткими вещественными числами (32 бита), `L` — вывод длинными вещественными числами (64 бита), `T` — вывод 10-байтовыми блоками;
- *`address`* — адрес, с какого предполагается вывод дампа. Можно указывать регистр, откуда этот адрес будет взят;
- *`length`* — количество выводимых в листинге байтов (длина). По умолчанию это значение равно 128.

Вывод осуществляется в окно данных. В случае если это окно отсутствует, дамп выводится в командное окно.

❒ `E` — команда редактирования памяти. Формат команды:

```
E[size] [address [data_list]]
```

Параметры:

- *`size`* — имеет тот смысл и значение, что и для команды `D`;
- *`address`* — определяет адрес редактируемой области;

- *data_list* — при отсутствии данного параметра курсор переходит в окно данных, где вы можете непосредственно отредактировать ячейку памяти. В качестве данного параметра выступают данные, которые помещаются в ячейки памяти, начиная с указанного адреса. Формат данных должен соответствовать параметру *size*. Если значений несколько, то они должны отделяться друг от друга запятыми.

Пример использования команды: `EB EBX 33,34,35` — по этой команде, начиная с адреса, который находится в регистре `EBX`, в три ячейки памяти будут помещены, соответственно, значения 33, 34, 35.

- `PEEK` — команда чтения непосредственно из физической памяти. Формат команды:

  `PEEK[`*size*`]` *address*

  Параметры:

  - *size* — размер ячейки памяти, принимает значения: `B` — байт, `W` — слово, `D` — двойное слово;
  - *address* — адрес, откуда производится чтение.

- `POKE` — команда записи непосредственно в физическую память. Формат команды:

  `POKE[`*size*`]` *address value*

  Параметры:

  - *size* — имеет такой же смысл, как и для команды `PEEK`;
  - *address* — физический адрес, куда осуществляется запись;
  - *value* — записываемое в физическую память значение.

- `PAGEIN` — загрузка отсутствующей страницы в физическую память. Формат команды: `PAGEIN` *address*. Параметром данной команды является виртуальный адрес страницы.
- `WATCH` — с помощью данной команды задается выражение, которое затем будет отслеживаться в окне слежения. Пример использования команды: `WATCH ds:eax`. Таким образом, будут отслеживаться данные, адрес которых находится в регистре `EAX`.
- `FORMAT` — с помощью данной команды можно изменить формат вывода в окне данных. Команда не имеет параметров. Она просто циклически (по кругу) переводит содержимое окна из одного формата в другой.
- `DATA` — с помощью данной команды можно создавать еще окна, для просмотра данных. В качестве параметра для данной команды можно использовать номер окна от 0 до 3.

- `A` — команда для ввода по указанному адресу ассемблерной команды. Формат команды: `A [`*`address`*`]`. Единственным параметром команды является адрес, куда будет помещена вводимая вами ассемблерная команда. Если адрес не указывать, то берется текущий адрес из области кода. При выполнении команды в командном окне появляется подсказка (адрес), после чего вы можете ввести ассемблерную команду.
- `S` — команда поиска данных. Формат команды:

  `S [-acu] [`*`address`*` L `*`length data_list`*`]`

  Параметры:

  - `c` — поиск без учета регистра;
  - `u` — поиск в формате Unicode;
  - `a` — поиск в формате ASCII;
  - *`address`* — начальный адрес поиска;
  - *`length`* — размер охватываемой поиском области памяти;
  - *`data_list`* — перечень данных для поиска, разделенных друг от друга запятыми или пробелами.

  Данная команда предназначена для поиска нужных данных. В случае их обнаружения они будут отображены в окне данных, а в командном окне появится соответствующее сообщение с указанием адреса расположения. Для продолжения поиска следует ввести эту команду без параметров. Пример: `S ds:eax L 2000 20` — поиск байта `20h` в области длиной в `2000h`, которая начинается с адреса, хранящегося в регистре `EAX`.
- `F` — команда заполнения области памяти. Формат команды:

  `F `*`address`*` L `*`length data_list`*

  Параметры:

  - *`address`* — начальный адрес;
  - *`length`* — длина заполняемой области;
  - *`data_list`* — данные, которые будут помещены, начиная с указанного адреса, разделенные запятыми либо пробелами.

  Эта команда помещает данные, указанные в *`data_list`*, начиная с заданного адреса. Если *`length`* больше длины данных — они будут циклически повторены до достижения размера *`length`*. Пример: `F ds:eax L 100 "W"` — область, которая начинается по адресу `DS:EAX` и имеет длину `100H`, будет заполнена символами `W`.
- `M` — команда перемещения данных. Формат команды:

  `M `*`address1`*` L `*`length address2`*

Параметры:

- *address1* — адрес, откуда будут переноситься данные;
- *length* — длина переносимых данных;
- *address2* — адрес, куда будут переноситься данные.

Пример команды: `M ds: eax L 1000 ds:ebx` — по данной команде `1000h` байтов будет перенесено из адреса, на который указывает `EAX`, в область с адресом, который хранится в регистре `EBX`.

- `C` — команда сравнения двух блоков данных. Формат команды:

  `C` *`address1`* `L` *`length address2`*

  Параметры:

  - *address1* — адрес первого сравниваемого блока;
  - *length* — длина сравниваемых данных;
  - *address2* — адрес второго блока данных.

  С помощью данной команды можно сравнить два блока данных. Если будет обнаружено неравенство, то в командном окне будут выведены полные адреса этих байтов и их значения. Пример: `C ds:100 L 10 ds:200` — будет произведено сравнение `10h` байтов.

- `HS` — данная команда может быть использована для поиска в командном буфере. Формат команды: `HS [+|-]` *`string`*. Знаки `+` или `-` определяют, соответственно, нисходящий (сверху вниз) и восходящий (снизу вверх) поиск. Далее указывается строка для поиска. Для продолжения поиска следует использовать команду без параметров.
- Команда `.` — точка. Если окно кода видимо, то данная команда делает инструкцию по адресу `CS:EIP` видимой и подсвечивает ее.

Команды управления точками прерывания (останова)

Точки останова или точки прерывания являются важнейшим механизмом отладки приложений. SoftICE присваивает каждой точке прерывания номер от 0 до 255. Таким образом, всего одновременно может существовать 256 точек прерывания. С помощью этого номера можно управлять точками прерывания: удалять и включать/выключать. Количество точек останова на обращение к памяти и портам ввода/вывода в сумме не должно превышать 4.

Типы точек прерывания

Перечислим основные точки прерывания, которые поддерживает отладчик SoftICE.

- Прерывание по исполняемым командам. В этом прерывании можно задавать имя, при появлении которого в исполняемом коде должна произойти

остановка выполнения кода. В частности вы можете установить точку останова на вызов какой-либо функции API.

- Прерывание на обращение к памяти. По данному прерыванию отладчик следит за обращением по определенному адресу памяти.
- Точки останова прерываний. Отладчик будет отслеживать прерывания, происходящие в операционной системе, путем модификации таблицы IDT.
- Прерывания на команды ввода/вывода. Отладчик отслеживает инструкции `IN`/`OUT`.
- Прерывания на сообщения Windows. При этом надо знать дескриптор окна, куда должно прийти данное сообщение.

Возможности точек прерывания

При работе с точками прерывания можно использовать условные конструкции. Тогда точка останова сработает только тогда, когда указанное условие будет выполнено. В частности, с помощью условия можно определить, для какого процесса будет срабатывать данная точка останова. Типичный пример такого условия `if(pid==0x058)` — условие того, что идентификатор процесса должен быть равен значению `0x058`. Этим условием нам придется пользоваться постоянно, поскольку мы будем отлаживать конкретные запущенные приложения.

При помощи оператора `do` можно задать команды, которые будут выполняться, если сработает точка останова. В общем случае формат команды выполнения действий имеет вид: `do "`*`команда 1;команда 2;`*`..."`. В качестве команд могут выступать и обычные команды отладчика или макрокоманды.

Далее при описании команд, применяемых при работе с точками останова, будут использоваться следующие обозначения.

- `size` — определяет размер ячейки, на которую будет устанавливаться точка прерывания. Может принимать значение: `B` — байт, `W` — слово, `D` — двойное слово.
- Параметр `[R|W|RW|X]` определяет тип доступа к ячейке памяти и порту ввода/вывода, который будет отслеживаться. `R` — чтение из ячейки (порта), `W` — запись в ячейку (порт), `RW` — чтение и запись в ячейку (порт), `X` — выполнение команды, занимающей данную ячейку памяти.
- `Reg_deb` — здесь можно указать, какой регистр отладки следует использовать (`D0`—`D3`). Как правило, это не делают, т. к. отладчик выбирает нужный регистр.
- `[IF` *`cond`*`]` — здесь нужно указать условие, которое должно выполниться, чтобы было возможно прерывание по данной точке останова.

- `[DO comm]` — можно указать команду или группу команд, которые будут выполняться при прерывании в данной точке.

Команды установки точек прерывания

- `BPM` — с помощью данной команды можно установить точку прерывания на ячейку памяти. Формат команды:

  `BPM[size] addr [R|W|RW|X] [reg_deb] [IF cond] [DO comm]`

  Параметр `addr` определяет адрес ячейки. Адрес можно указать явно или посредством регистров, например `ds:eax`.

- `BPIO` — данная точка останова устанавливается на ввод/вывод в указанный порт. Формат команды:

  `BPIO [R|W|RW] [deb_reg] [IF cond] [DO comm]`

  Отладчик будет отслеживать все команды ввода/вывода в указанный порт.

- `BPINT` — данная команда используется для установки точки останова на прерывание. Точка останова срабатывает только в том случае, если прерывание срабатывает через IDT (таблицу дескрипторов прерываний). Формат команды:

  `BPINT int_number [IF cond] [DO comm]`

  Здесь `int_number` — номер отслеживаемого прерывания. При срабатывании точки останова первой инструкцией будет первая команда обработчика прерываний.

- `BPX` — эта команда устанавливает точку останова на выполнение. Например, на выполнение какой-либо функции API. Формат команды:

  `BPX имя [IF cond][DO comm]`

  Здесь `имя` — некоторое имя. Пример: `BPX MessageBoxW`. Команда `BPX`, не содержащая параметров, устанавливает точку останова на текущую команду, но для этого следует перейти в окно кода отладчика.

- `BMSG` — команда предназначена для установки точки прерывания на сообщения, приходящие для конкретного окна в определенном диапазоне. Формат команды:

  `BMSG hWnd [L] [beg_mes [end_mes]][IF cond] [DO comm]`

  Здесь:

  - `hWnd` — дескриптор окна;
  - `[L]` — при установке этого параметра сообщение будет просто отображено в буфере (окне) команд, а сам отладчик не будет активизирован;
  - `[beg_mes]` — первое сообщение диапазона сообщений. Может быть задан как числовым, так и символьным обозначением сообщения;

- *[end_mes]* — последнее сообщение диапазона (если речь идет именно о диапазоне, а не об одном сообщении). Если данный параметр отсутствует, то отлавливается лишь сообщение, заданное параметром *beg_mes*.

Если сообщения в команде не указывать, то точка останова накладывается на все сообщения данного окна. Пример использования команды: `BMSG 01001F WM_PAINT` — перехват сообщения `WM_PAINT` для окна с дескриптором `01001F`.

- `BSTAT` — данная команда служит для выдачи статистической информации по заданной точке останова. В качестве параметра данной команды следует указать номер точки прерывания. В частности, будет выдана величина `Popups` — количество раз, когда данная точка прерывания вызывала вызов окна SoftICE, `Breaks` — количество срабатываний точки останова и т. д.

Команды манипулирования точками прерывания

- `BPE` — команда редактирования точки останова. Параметром данной команды служит номер точки останова.
- `BPT` — данная команда вызывает в командную строку шаблон создания точки останова с заданным номером. Отличие от предыдущей команды заключается в том, что с помощью данной команды создается еще одна точка останова, а не редактируется уже существующая.
- `BL` — данная команда выдает список точек останова — номер и шаблон создания.
- `BC` — команда удаления точки останова. Параметром данной команды может служить номер точки останова или список номеров (через запятую или пробел), подлежащих удалению. Если в качестве параметра использовать знак `*`, то будут удалены все точки останова.
- `BD` — команда приостанавливает работу точек останова. В качестве параметра данной команды может быть список точек останова (номера через запятую или пробел) или знак `*`.
- `BE` — команда возобновляет работу точек останова. В качестве параметра данной команды может быть список точек останова (номера через запятую или пробел) или знак `*`.
- `BH` — выдает список точек останова, которые использовались в данной и предыдущей сессии работы отладчика. Продвигаясь по появившемуся списку, можно с помощью клавиши <Insert> выбрать точку останова, которую вы хотите использовать сейчас. Клавиша <Enter> используется для установки всех выбранных точек останова. Отладчик помнит последние 32 точки останова.

Команды трассировки

- `X` — выход из окна SoftICE и возвращение управления программе, прерванной вызовом SoftICE. Равносильно нажатию клавиши <F5> или комбинации клавиш <Ctrl>+<D>.
- `G` — команда сообщает отладчику, что необходимо выполнить отлаживаемое приложение. Формат команды:

  `G [=`*address1*`] [`*address2*`]`

  Параметры:

  - *address1* — адрес, с которого должно начаться выполнение. Если данный адрес не указан, то выполнение начнется с текущего адреса (`CS:EIP`);
  - *address2* — конечный адрес выполнения. Если данный адрес не указан, то выполнение будет происходить до тех пор, пока не встретится точка останова или не будет выполнен вызов окна SoftICE.

  Команда `G` без параметров равносильна команде `X`. Команда `G @SS:EBP` равносильна нажатию клавиши <F11> — перейти в вызывающую функцию.
- `T` — команда пошагового выполнения отлаживаемого кода. Формат команды:

  `T [=`*address*`] [`*count*`]`

  Параметры:

  - *address* — начальный адрес трассировки. Если данный адрес не указан, то выполнение начинается с текущей команды;
  - *count* — указывает, сколько инструкций следует выполнить. Если параметр отсутствует, то выполняется одна команда.

  Команда без параметров равносильна нажатию клавиши <F8>. Пример команды `T`: `T CS:EIP-20 10` — будет выполнено `10` инструкций, начиная с адреса `CS:EIP-20`.
- `P` — данная команда — это выполнение инструкции с обходом вызова процедур, прерываний, а также строковых команд и циклов. Без параметров команда равносильна нажатию клавиши <F10>. Если присутствует опция `RET` (`P RET`), то SoftICE будет выполнять программу до обнаружения инструкции `RETN`/`RETF`, причем остановка будет там, куда произойдет переход с помощью этих команд. Таким образом, с параметром команда равносильна нажатию клавиши <F12>.
- `HERE` — данная команда равносильна нажатию клавиши <F7> — выполнить программу с адреса `CS:EIP` и до текущего положения курсора в окне кода.

- `EXIT` — считается устаревшей командой. Фактически равносильна команде `X`. Следует избегать использования этой команды.
- `GENINT` — передача управления прерыванию. Формат команды:

  `GENINT [nmi|int1|int3|`*`number`*`]`

  Параметры:

  - `nmi` — вызов немаскируемого прерывания;
  - `int1` — вызов прерывания с номером 1;
  - `int3` — вызов прерывания с номером 3;
  - *`number`* — вызов прерывания с номером от 0 до 5F.

  Использовать данную команду следует очень осторожно. Вы должны быть уверены, что обработчик прерывания существует, в противном случае команда вызовет зависание системы.
- `HBOOT` — команда осуществляет сброс (перезагрузку) компьютерной системы.
- `I1HERE` — имеет две формы: `I1HERE on` — включение режима, окно отладчика SoftICE будет вызываться каждый раз, когда будет происходить прерывание с номером 1. `I1HERE off` — выключение режима.
- `I3HERE` — имеет две формы: `I3HERE on` — включение режима, окно SoftICE будет вызываться каждый раз, когда будет происходить прерывание с номером 3. `I3HERE off` — выключение режима.
- `ZAP` — данная команда заменяет вызовы прерываний с номерами 1 и 3 на инструкции `NOP`.

Основные информационные команды

- `GDT` — команда для отображения таблицы GDT. Формат команды:

  `GDT [`*`selector`*`|`*`address`*`]`

  Параметры:

  - *`selector`* — селектор в таблице GDT;
  - *`address`* — адрес сегмента.

  Если не указывать параметров, то будет отображено содержимое всей таблицы GDT.
- `LDT` — команда для отображения таблицы LDT. Формат команды:

  `LDT [`*`selector`*`|`*`table_selector`*`]`

  Параметры:

  - *`selector`* — селектор в LDT;

- *table_selector* — селектор LDT в GDT.

 Команда без параметров выдает всю таблицу LDT.

- IDT — команда для отображения содержимого таблицы прерываний. Формат команды:

 ```
 IDT [number|address]
 ```

 Параметры:

 - *number* — номер прерывания, информацию о котором из таблицы IDT следует отобразить;
 - *address* — адрес обработчика прерываний (селектор:смещение), информацию о котором из таблицы IDT следует отобразить.

 Без параметров команда выводит текущее содержимое все таблицы IDT.

- TSS — по данной команде в командном окне будет выведено содержание сегмента TSS. Параметром команды является селектор в GTD, указывающий на TSS. Если команду запустить без указания селектора, то будет показано содержание текущего TSS, селектор которого находится в регистре задач TR.

- CPU — на данную команду выдается полный список регистров процессора и их содержимое.

- PCI — команда выводит в командном окне информацию обо всех PCI-устройствах, имеющихся в системе.

- MOD — по данной команде в окно команд выдается список всех подключенных модулей Windows. В командной строке можно указать первые буквы имени модуля, тогда будет выдан список модулей, имена которых начинаются с указанного префикса.

- HEAP32 — выдает список системных и созданных приложениями куч (heap) памяти. Формат команды:

 ```
 HEAP32 [hheap|name]
 ```

 Параметры:

 - *hheap* — дескриптор кучи, возвращаемый функцией CreateHeap;
 - *name* — имя задачи.

 Команда выдает: базовый адрес кучи, максимальный размер, количество килобайт используемой памяти, количество сегментов в куче, тип кучи, владельца кучи. В отсутствие параметров команда выдает список всех куч.

- TASK — по данной команде в командном окне будет отображен весь список задач и дополнительная информация о них. Возле активной задачи будет символ *.

Команда может быть полезна в случае сбоя в системе для определения задачи, вызвавшей его.

- `NTCALL` — команда выдает список системных сервисов, функционирующих на уровне ядра (кольцо 0).
- `WMSG` — выдает в командное окно список сообщений Windows и их номера. Формат команды:

  `WMSG [`*`partial_name`*`] [`*`number`*`]`

  Параметры:

  - *`partial_name`* — полное или частичное название сообщения;
  - *`number`* — номер сообщения Windows.

  Команда без параметров выдает список всех известных отладчику сообщений Windows. При наличии параметра *`partial_name`* выдаются все сообщения, соответствующие данному фрагменту имени сообщения. Если указан номер сообщения, то будет выдан номер и имя сообщения.
- `PAGE` — по данной команде будет выдана информация о страницах, начинающихся с данного виртуального адреса (виртуальный и физический адрес, атрибут, тип, виртуальный размер). Формат команды:

  `PAGE [`*`address`*`] [L `*`length`*`]`

  Параметры:

  - *`address`* — виртуальный адрес страниц;
  - *`length`* — количество выдаваемых страниц.

  Команда без параметров выдает список всех страниц.
- `PHYS` — данная команда вызывает отображение списка всех виртуальных адресов, соответствующих указанному физическому адресу. Данная команда используется только с параметром — физическим адресом.
- `STACK` — выдает информацию о структуре стека. Формат команды:

  `STACK [`*`thread`*` | `*`frame`*`]`

  Параметры:

  - *`thread`* — дескриптор или идентификатор потока;
  - *`frame`* — адрес стекового фрейма.

  Команда без параметров выдает информацию о текущем стеке, на основе адреса `SS:EBP`.
- `XFRAME` — выдает записанную в стек информацию об исключении. Параметром команды служит идентификатор потока или указатель на фрейм стека. Если параметр отсутствует, то отладчик использует текущий поток.

- `HWND` — команда выдает информацию о созданных в системе окнах. Формат команды:

  `HWND [-x] [-c] [`*`hwnd`*`|`*`desktop`*`|`*`process`*`|`*`thread`*`|`*`module`*`|`*`class`*`]`

  Параметры:

  - `-x` — вывод расширенной информации;
  - `-c` — заставляет отладчик выдавать иерархию окон;
  - *`hwnd`* — дескриптор окна или указатель на структуру окна;
  - *`desktop`* — дескриптор рабочего стола;
  - *`process`* — идентификатор процесса;
  - *`thread`* — идентификатор потока;
  - *`module`* — имя модуля;
  - *`class`* — имя зарегистрированного класса окон.

  Команда без параметров выдает информацию обо всех созданных на данный момент в системе окнах.

- `CLASS` — выдает информацию о классах окон. Формат команды:

  `CLASS [-x] [`*`process`*`] [`*`thread`*`] [`*`module`*`] [`*`class`*`]`

  Параметры:

  - `-x` — выдавать расширенную информацию о классах;
  - *`process`* — идентификатор процесса;
  - *`thread`* — идентификатор потока;
  - *`module`* — идентификатор или имя модуля;
  - *`class`* — имя зарегистрированного класса.

  Команда без параметров выдает список всех зарегистрированных классов текущего процесса.

- `THREAD` — команда используется для получения информации о потоках. Формат команды:

  `THREAD [-r|-x-u] [`*`thread`*`] [`*`process`*`]`

  Параметры:

  - `-r` — выдавать команду о регистрах потока;
  - `-x` — выдавать расширенную информацию о потоках;
  - `-u` — выдавать информацию о компонентах потока пользовательского уровня;
  - *`thread`* — идентификатор процесса;
  - *`process`* — идентификатор процесса.

- `ADDR` — используется для выдачи информации о существующих адресных контекстах (процессов) и установлении текущего контекста. Для установления текущего контекста параметром команды следует указать идентификатор, имя процесса или адрес. Можно также указать адрес информационного блока процесса (KPEB, Kernel Process Environment Block — блок описания процесса уровня ядра). Всю эту информацию можно получить, если использовать команду `ADDR` без параметров.
- `MAP32` — выдает список загруженных 32-битных модулей и дополнительную информацию о них. Формат команды:

  `MAP32 [-u|-s] [`*`name|handle|address`*`]`

  Параметры:

  - `-u` — показать только модули, загруженные в часть оперативной памяти, которую можно использовать для хранения программ пользователя;
  - `-s` — показать только модули, загруженные в часть оперативной памяти, отведенную под операционную систему и требуемые ей средства;
  - *`name`* — имя модуля;
  - *`handle`* — адрес модуля;
  - *`address`* — адрес, принадлежащий модулю.

  Команда без параметров выдаст список всех загруженных 32-битных модулей и дополнительную информацию о них.
- `PROC` — команда предназначена для получения информации о процессе. Формат команды:

  `PROC [-xom] [`*`name`*`]`

  Параметры:

  - `-x` — показать расширенную информацию о каждой ветви;
  - `-o` — показать расширенную информацию о каждом объекте;
  - `-m` — показать информацию об использовании объектом памяти;
  - *`name`* — имя задачи, имя процесса, дескриптор процесса, идентификатор процесса, имя потока, идентификатор потока, дескриптор потока.

  Если не указано имя объекта, то выводится информация обо всех процессах системы.
- `QUERY` — команда предназначена для вывода карты виртуальной памяти процессов. Формат команды:

  `QUERY [-x] [`*`address`*`] [`*`name`*`]`

  Параметры:

  - `-x` — показать имена процессов (и дополнительную информацию о них), которые занимают указанный виртуальный адрес;

- *address* — виртуальный адрес;
- *name* — имя процесса.

Без параметров команда выдает карту виртуальной памяти текущего процесса.

- WHAT — данная команда пытается интерпретировать указанный в ней параметр. Например, если параметр — это идентификатор процесса, то команда сообщает вам об этом, т. е. вы тем самым можете проверить подлинность идентификатора или дескриптора.
- OBJTAB — команда для получения информации о таблице объектов USER.
- FOBJ — выдает информацию о файловых объектах, существующих в настоящее время. Такие объекты создаются для каждого открытого файла.
- IRP — команда выдает информацию об IRP (IRP — I/O request packet, т. е. пакет запроса ввода/вывода).
- FIBER — выдает структуру данных для волокон. Эта структура данных, в частности, возвращается функцией CreateFiber.

Другие команды

- PAUSE — устанавливает два режима просмотра информации в командном окне. PAUSE on (по умолчанию) — информация выдается порциями, следующая порция появляется по нажатию любой клавиши. PAUSE off — информация выдается непрерывно.
- ? — команда вычисления выражения. Например: ? 34+90*2. Отладчик при этом выводит результат одновременно в шестнадцатеричной и десятичной системе счисления, а также в ASCII-формате.
- OPINFO — получить информацию об инструкции процессора. Например, по команде OPINFO add на экран будет выдана основная информация об инструкции процессора ADD:

```
ADD
      Integer addition: DEST <- DEST + SRC
      EFLAGS | OF DF IF SF ZF AF PF CF TF NT RF |
             | M        M  M  M  M  M           |
```

- ALTKEY — данная команда производит изменение комбинаций клавиш, используемых для активизации SoftICE. По умолчанию используется комбинация клавиш <Ctrl>+<D>. Если эту команду запустить без параметров, то SoftICE отобразит в командном окне текущую комбинацию. Примеры использования команды: ALTKEY Alt P, ALTKEY Ctrl Z — теперь окно SoftICE будет вызываться нажатием клавиш <Ctrl>+<Z>.

Операторы

В среде отладчика SoftICE в командах и определении условных точек останова можно использовать выражения. Для построения выражений можно использовать операторы. Рассмотрим перечень используемых в отладчике операторов.

Операторы адресации

- Команда (оператор) `.` — точка. Если окно кода видимо, то данная команда делает инструкцию по адресу `CS:EIP` видимой и подсвечивает ее. Точку можно использовать в выражениях.
- `*` — данный оператор используется для задания адреса, на который указывает это выражение. Например, `*(EAX)` означает содержимое памяти, на которое указывает регистр `EAX`.
- `->` — с помощью данного оператора, так же как и с помощью оператора "звездочка", можно получить содержимое по адресу, на который указывает это выражение. Например, если вам известен адрес процедуры окна, который в частности можно получить при помощи команды `HWND`, то можно использовать следующую точку останова на сообщение `WM_PAINT`: `BPX 6BDFE003 IF (ESP->8) ==WM_PAINT`.
- `@` — фактически эквивалентен оператору "звездочка".

Математические операторы

- Унарный и бинарный операторы `+` (плюс). Например, `+100` или `EBX+ESI`.
- Унарный и бинарный операторы `-` (минус). Например, `-100` или `EAX-8`.
- Бинарный оператор `*` (умножение). Например, `EBX*4`.
- Бинарный оператор `/` (деление). Например, `(EAX+EBX)/2`.
- Бинарный оператор `%` (деление по модулю). Например, `EBX%3`.
- Оператор логического сдвига влево `<<`.
- Оператор логического сдвига вправо `>>`.

Побитовые операторы

- Побитовый оператор "И" — `&`.
- Побитовый оператор "ИЛИ" — `|`.
- Побитовый оператор "исключающее ИЛИ" — `^`.
- Побитовый оператор "инверсия" или "НЕ" — `~`.

Логические операторы

- Логическое отрицание ("НЕ") — `!`. Например, `!EBX`.
- Логическое "И" — `&&`. Например, `EAX&&EBX`.
- Логическое "ИЛИ" — `||`. Например, `EAX||FF`.
- Условие равенства `==`.
- Условие неравенства `!=`.
- Меньше `<`.
- Больше `>`.
- Меньше или равно `<=`.
- Больше или равно `>=`.

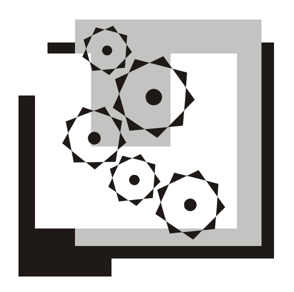
Встроенные функции SoftICE

Дизассемблер имеет ряд собственных встроенных функций. Вот перечень основных функций:

- `Byte` — возвращает младший байт выражения;
- `Word` — возвращает младшее слово выражения;
- `Dword` — возвращает двойное слово (расширяет байт или слово);
- `HiByte` — возвращает старший байт (слова или двойного слова);
- `HiWord` — возвращает старшее слово;
- `Sword` — преобразует байт в слово со знаком;
- `Long` — преобразует байт или слово в длинное целое;
- `WSTR` — показать строку в формате Unicode;
- `Flat` — преобразовать адрес с селектором (логический адрес) в линейный адрес плоской модели памяти;
- текущее содержимое регистров может быть найдено при помощи функций, с названием, соответствующим названию регистра. Например, `EAX`, `EBX`, `EDX` и т. д.;
- `CFL` — возвращает флаг переноса;
- `PFL` — возвращает флаг четности;
- `AFL` — возвращает флаг вспомогательного перехода;
- `ZFL` — возвращает флаг нуля;
- `SFL` — возвращает знаковый флаг;
- `OFL` — возвращает флаг переполнения;

- RFL — возвращает флаг возобновления;
- TFL — возвращает флаг трассировки;
- DFL — возвращает флаг направления;
- IFL — возвращает флаг разрешения прерывания;
- NTFL — возвращает флаг вложенной задачи;
- IOPL — возвращает уровень привилегий ввода/вывода;
- VMFL — возвращает флаг режима виртуального процессора;
- IRQL — возвращает текущий IRQ;
- DataAddr — возвращает начальный адрес блока данных, отображаемых в окне данных;
- CodeAddr — возвращает адрес первой инструкции, отображаемой в окне кода;
- Eaddr — возвращает эффективный адрес текущей инструкции, если таковая есть;
- Evalue — возвращает значение по текущему эффективному адресу;
- Process — возвращает блок среды активного процесса;
- Thread — возвращает блок среды активного потока;
- PID — идентификатор активного процесса;
- TID — идентификатор активного потока;
- BPCount — возвращает количество попаданий в точку прерывания, при которых значение условного выражения равно TRUE;
- BPTotal — возвращает общее количество попаданий в точку прерывания;
- BPMiss — возвращает количество попаданий в точку прерывания, при которых условие прерывания не было выполнено и окно SoftICE не активизировалось;
- BPLog — сохраняет в буфере информацию о попадании в точку прерывания;
- BPIndex — возвращает номер текущей точки останова в общем списке точек.

Если вы предваряете символом подчеркивания имя функции в каком-либо выражении, то дизассемблер вычисляет значение функции на данный момент и использует это значение в дальнейших вычислениях. Например, _PID, _TID, _EAX и т. д.



Глава 4.4

Основы анализа кода программ

Большинство исполняемых модулей были написаны на языках высокого уровня и не содержат отладочной информации. Однако анализировать код программы приходится и в этом случае. Для того чтобы ускорить процесс анализа, необходимо знать или, по крайней мере, иметь перед глазами некоторые стандартные ассемблерные структуры, соответствующие определенным структурам языков высокого уровня. Замечу, что речь, разумеется, будет идти о 32-битных приложениях. Для более подробного знакомства с вопросами исследования исполняемого кода можно порекомендовать книгу автора "Ассемблер и дизассемблирование" (см. [27]).

Переменные и константы

Современные компиляторы довольно эффективно оптимизируют исходный код, поэтому не всегда просто разобраться, где какая переменная работает. Сложность в первую очередь заключается в том, что для хранения части переменных, насколько это возможно, компилятор использует регистры. Если ему не хватит регистров, он начнет использовать память.

Для примера я взял простую консольную программу, написанную на Borland C++. В текстовом варианте программа занимает полтора десятка строк, тогда как exe-файл имеет размер более 50 Кбайт. Впрочем, размер исполняемых файлов давно уже никого не удивляет. Интересно другое: корректно справился с задачей, т. е. корректно выявил точку входа — метку `_main`, только один дизассемблер — IDA Pro. То есть, конечно, реально работающий участок программы дизассемблировали все, но выявить, как происходит переход на участок, смог только упомянутый мной дизассемблер. Приятно также и то, что аккуратно была распознана функция `_printf`. В листинге 4.4.1 показан фрагмент дизассемблированной программы, соответствующей основной

процедуре `main`. С другой стороны, в данном случае нет никаких видимых возможностей быстрого поиска данного фрагмента в отладчике. Отсюда наглядно можно понять полезность совместного использования отладчика и дизассемблера.

Листинг 4.4.1. Функция `main` консольного приложения. Дизассемблирование с помощью IDA Pro программы на языке C

```
CODE:00401108 _main  proc near ; DATA XREF: DATA:0040B044
CODE:00401108
CODE:00401108 argc   = dword ptr 8
CODE:00401108 argv   = dword ptr 0Ch
CODE:00401108 envp   = dword ptr 10h
CODE:00401108
CODE:00401108        push  ebp
CODE:00401109        mov  ebp, esp
CODE:0040110B        push  ebx
CODE:0040110C        mov  edx, offset unk_40D42C
CODE:00401111        xor  eax, eax
CODE:00401113
CODE:00401113 loc_401113: ; CODE XREF: _main+22
CODE:00401113        mov  ecx, 1Fh
CODE:00401118        sub  ecx, eax
CODE:0040111A        mov  ebx, ds:off_40B074
CODE:00401120        mov  cl, [ebx+ecx]
CODE:00401123        mov  [edx+eax], cl
CODE:00401126        inc  eax
CODE:00401127        cmp  eax, 21h
CODE:0040112A        jl   short loc_401113
CODE:0040112C        mov  byte ptr [edx+20h], 0
CODE:00401130        push  edx       ; char
CODE:00401131        push  offset aS ;__va_args
CODE:00401136        call  _printf
CODE:0040113B        add  esp, 8
CODE:0040113E        pop  ebx
CODE:0040113F        pop  ebp
CODE:00401140        retn
CODE:00401140 _main  endp
```

Попытаемся теперь понять, какая программа на C была исходным источником данного фрагмента. Начнем с рассмотрения стандартных структур. Собственно, налицо только одна структура — цикл. Команда

```
CODE:0040112A        jl short loc_401113
```

и является ключевой в организации цикла. Команда же `inc eax`, очевидно, инкрементирует параметр цикла. Таким образом, в `EAX` хранится некая пере-

менная, играющая роль параметра цикла. Назовем эту переменную `i`. Вышесказанное подтверждается и наличием команды `xor eax,eax` перед началом цикла. Команда эта, разумеется, эквивалентна просто `i=0`. Команда `inc eax` означает `i++`. Попытаемся выявить еще переменные. Обратим внимание на команду

```
CODE:0040110C         mov edx, offset unk_40D42C
```

Команда весьма примечательна, т. к. в регистр `EDX` помещается некий адрес, адрес чего-то. Проследим далее, как используется регистр `EDX`. Команда

```
CODE:00401123         mov [edx+eax], cl
```

а также отсутствие в цикле других команд с регистром `EDX` убеждает нас, что `EDX` играет роль указателя на строку, массив или запись. Это нам предстоит сейчас выяснить. Тут примечательны две команды:

```
CODE:0040112C         mov  byte ptr [edx+20h],0
```

и

```
CODE:00401130         push  edx     ; char
```

Первая команда убеждает нас, что `EDX` указывает на строку, т. к. именно строка должна содержать в конце символ 0. Вторая команда осуществляет передачу второго параметра в функцию `printf`. Исходя из этого, а также комментария IDA Pro (отладчик раньше нас понял, что это такое), заключаем, что `EDX` представляет собой указатель на некую строку. Заметьте, что мы не обратились к просмотру блока данных, что, несомненно, ускорило бы наше исследование. Обозначим этот указатель как `s1`. В этой связи выражение `[edx+eax]` можно трактовать как `s1[i]` или как `*(s1+i)`.

Рассмотрим теперь команду

```
CODE:0040111A         mov  ebx, ds:off_40B074
```

несомненно, означающую, что и регистр `EBX` также представляет собой указатель на строку (будем обозначать ее `s2`), подтверждением этому также служат последующие строки, означающие переброску символов из строки `s2` в строку `s1`. Вот об этом следует поговорить подробнее.

Что означает последовательность команд

```
CODE:00401113         mov  ecx, 1Fh
CODE:00401118         sub  ecx, eax
```

Эта последовательность может означать только одно: на каждом шаге цикла в `ECX` будут находиться числа от `1FH` (31) до `0` (последним значением регистра `EAX`, т. е. переменной `i`, участвующим в команде `sub ecx,eax`, будет `1FH`). Поскольку в формировании содержимого регистра `ECX` участвует еще команда `mov cx,1FH`, логично предположить, что в регистре `ECX` перед операцией пере-

качки символов из одной строки в другую всегда будет находиться число 1FH-I или 31-i. Выражение [ebx+ecx] тогда будет эквивалентно s2[31-i] или *(s2+31-i).

В результате получаем, что строки

```
CODE:00401120         mov  cl, [ebx+ecx]
CODE:00401123         mov  [edx+eax], cl
```

по сути, можно заменить на s1[i]=s2[31-i].

Думаю, что теперь мы готовы рассмотреть целый фрагмент:

```
CODE:00401111         xor  eax, eax
CODE:00401113
CODE:00401113 loc_401113: ; CODE XREF: _main+22
CODE:00401113         mov  ecx, 1Fh
CODE:00401118         sub  ecx, eax
CODE:0040111A         mov  ebx, ds:off_40B074
CODE:00401120         mov  cl, [ebx+ecx]
CODE:00401123         mov  [edx+eax], cl
CODE:00401126         inc  eax
CODE:00401127         cmp  eax, 21h
CODE:0040112A         jl   short loc_401113
```

Надеюсь, что не ошибусь, написав на языке С следующий фрагмент:

```
i=0;
do {
  s1[i]=s2[31-i];
  i++;
} while(i>0x20)
```

Все бы хорошо, но непонятным остается наличие в цикле строки

```
CODE:0040111A   mov ebx, ds:off_40B074
```

Почему бы не поместить ее, например, перед циклом? Ну что ж, оставим это на совести компилятора. Что касается того, что цикл этот должен быть обязательно циклом DO, то это совсем не обязательно. В данном случае количество шагов цикла задано явно, и можно сделать один условный переход в конце цикла. Другими словами, приведенная выше структура могла быть результатом оптимизационного преобразования цикла while.

Можно задаться и еще одним вопросом: где хранятся строки s1 и s2? В этом можно разобраться достаточно быстро. Main является самой обычной процедурой, и если бы строки являлись локальными переменными, то для них была бы зарезервирована область в стеке процедуры путем вставки команды SUB ESP,N (или ADD ESP,-N). Таким образом, строковые переменные s1 и s2 являются глобальными. Интересно, что другие переменные, хотя они тоже

локальные, хранятся в регистрах, в полном соответствии с принципом, что по мере возможности переменные следует хранить в регистрах.[1]

Итак, окончательный результат наших изысканий может быть представлен в листинге 4.4.2.

Листинг 4.4.2. Окончательный вариант программы на C

```
char s1[32];
char * s2="абвгдежзийклмнопрстуфхцчшщыьъэюя";
void main()
{
  int i;
  i=0;
  do {
    s1[i]=s2[31-i];
    i++;
  } while(i>32);
  s1[32]=0;
  printf("%s\n",s1);
}
```

Ну, уж чтобы совсем покончить с рассматриваемым примером, отмечу, что переменная `s1` является неинициализированной, а переменная `s2` — инициализированной. Инициализированные переменные хранятся в секции `DATA`, неинициализированные — в секции `BSS` (для компилятора Borland C++).

Современные компиляторы допускают использование 64-битных целых чисел. Сам ассемблерный код при этом несколько усложняется. Впрочем, не так сильно. Надо иметь в виду, что 64-битная переменная хранится в двух смежных 32-битных блоках. Причем старшая часть переменной имеет больший адрес. Для регистрового манипулирования такой переменной обычно используется пара регистров `EDX:EAX`, соответственно, для старшей и младшей частей. Поэтому, например, такие строки как

```
MOV DWORD PTR [adr],EAX
MOV DWORD PTR [adr+4],EDX
```

должны сразу подсказать вам, что вы имеете дело с 64-битной переменной.

[1] В старом C для того чтобы для хранения переменных компилятор использовал регистры, переменные следует объявлять, используя модификатор `register`.

Управляющие структуры языка C

В данном разделе мы рассмотрим некоторые классические управляющие структуры языка C и их отображение в язык ассемблера при трансляции.

Условные конструкции

- Неполная условная конструкция.

```
if(простое условие)
{
  ...
}
```

Если условие простое, например `i==1`, то оно, разумеется, заменяется такой возможной последовательностью:

```
      CMP EAX,1
      JNZ L1
      ...
L1:
```

- Полная условная конструкция.

```
if(простое условие)
{
  ...
} else
{
  ...
}
```

Она заменяется на последовательность:

```
    CMP EAX,1
    JNZ L1
    ...
    JMP L2
L1:
    ...
L2:
```

Вложенные условные конструкции

Здесь все достаточно очевидно.

```
    CMP EAX,1
    JNZ L1
    CMP EBX,2
    JNZ L1
    ...
L1:
```

Что, конечно, равносильно одному составному условию, связанному союзом "И". Союз "ИЛИ", как известно, заменяется проверкой условий в блоке `else`.

Оператор *switch* или оператор выбора

Оператор `switch` весьма часто употребляется в функциях окон. Хорошее знание его ассемблерной структуры поможет вам легче отыскивать эти функции в море ассемблерного кода.

```
switch(i)
{
  case 1:
    ...
    break;
  case 3:
    ...
    break;
  case 5:
    ...
    break;
}
```

А вот соответствующий данной структуре ассемблерный код:

```
    DEC EAX
    JZ  L1
    SUB EAX,2
    JZ  L2
    SUB EAX,2
    JZ  L3
    JMP L4
L1:
    ...
    JMP L4
L2:
    ...
    JMP L4
L3:
    ...
L4:
```

Структура, как видите, интересная. Такой подход позволяет наилучшим образом оптимизировать проверку большого количества условий.

В действительности оператор выбора может кодироваться и другим способом. Вот еще один возможный вариант представления оператора выбора:

```
CMP EAX,10
JE  L1
```

```
CMP EAX,5
JE  L2
CMP EAX,11
JE  L3
...
```

Циклы

С организацией цикла в языке C мы уже познакомились в этой главе. Отмечу две наиболее часто практикуемые структуры.

- 1-я структура:

```
L1:
    ...
    CMP EAX,10
    JL  L1
```

- 2-я структура:

```
L2:
    CMP EAX,10
    JA  L1
    ...
    JMP L2
L1:
```

Первая структура соответствует следующему фрагменту:

```
int i;
...
do {
  ...
}
while (i < 10);
```

или на Паскале:

```
var i: integer;
. . .
repeat
. . .
until (i>=10);
```

Вторая структура соответствует фрагменту следующего вида:

```
unsigned int i;
...
while (i <= 10) {
  ...
}
```

или на Паскале:

```
var i:integer;
. . .
```

```
while (i<=10) do
begin
...
end;
```

Первая структура может быть несколько видоизменена и принять следующий вид:

```
      JMP L2
L1:
      ...
      CMP EAX,10
L2:
      JL  L1
```

Как по мановению волшебной палочки, в последней структуре цикл "до" превращается в цикл "пока".

В своем рассмотрении я не упомянул цикл `for` — этого "монстра" языка С, но, в принципе, это все тот же цикл "пока".

Локальные переменные

Чаще всего нам приходится работать с локальными переменными. С глобальными переменными мы уже встречались, они хранятся во вполне определенных секциях. Хороший дизассемблер их легко локализует, при условии распознавания ссылок на них. С локальными переменными часто разбираться гораздо сложнее. Особенно это касается записей или массивов. Хороший дизассемблер значительно упростит задачу. Рассмотрим, например, фрагмент из листинга 4.4.3, взятый из IDA Pro.

Листинг 4.4.3. Пример задания двух локальных массивов. Взят из отладчика IDA Pro

```
CODE:00401108 _main proc near ; DATA XREF: DATA:0040B044
CODE:00401108
CODE:00401108 var_54 = dword ptr -54h
CODE:00401108 var_28 = byte ptr -28h
CODE:00401108 argc   = dword ptr 8
CODE:00401108 argv   = dword ptr 0Ch
CODE:00401108 envp   = dword ptr 10h
CODE:00401108
CODE:00401108        push  ebp
CODE:00401109        mov  ebp, esp
CODE:0040110B        add  esp, 0FFFFFFACh
CODE:0040110E        push  ebx
CODE:0040110F        xor  ebx, ebx
```

Взгляните внимательно на фрагмент из листинга 4.4.3. Как видите, отладчик нам все прописал. Две переменные — `var_54` и `var_28` — являются, несомненно, массивами типа `DWORD`. Причем если на первый отводится `28h` байтов, т. е. 40 байтов, или 10 элементов массива, то на второй `54h-28h=2CH=44` байтов или 11 элементов массива. И всего, следовательно, под локальные переменные зарезервировано 84 байта. А что означает команда `add esp,0FFFFFFACH`? Но нас не обманешь! `0-0FFFFFFACH = 54H`, что в десятичном исчислении и есть 84. То есть в начале резервируется место под локальные переменные.

В связи с массивами хотелось бы упомянуть, что в C изначально практиковались два способа доступа к элементам массива: посредством индексов и посредством указателей. Другими словами, можно было написать: `a[i]=10` и `*(a+i)=10`. Причем вторая запись оказывалась более эффективной (см. по этому поводу книгу [1], *главу 15*). Разумеется, делать это можно и сейчас, но обе записи теперь в Borland C++ 5.0 и Microsoft C++ 6.0 приводят к совершенно одинаковым ассемблерным эквивалентам. Это весьма отрадно, значит, компиляторы действительно развиваются. Кстати, весьма поучительным было бы сравнение ассемблерного кода, производимого разными компиляторами. Мы не будем этим заниматься, замечу только, что мое весьма поверхностное сравнение компиляторов Borland C++ 5.0 и Visual C++ привело к выводу, что средства, разработанные фирмой Microsoft, несколько более эффективно оптимизируют транслируемый код.

Но вернемся опять к фрагменту из листинга 4.4.3. Посмотрим, как выглядит начало функции `main` в дизассемблере W32Dasm (листинг 4.4.4).

Листинг 4.4.4. Так выглядит фрагмент программы, представленный в листинге 4.4.3, в окне дизассемблера W32Dasm

```
:00401108 55          push ebp
:00401109 8BEC        mov  ebp, esp
:0040110B 83C4AC      add  esp, FFFFFFAC
:0040110E 53          push ebx
:0040110F 33DB        xor  ebx, ebx
```

Как видите, дизассемблер W32Dasm менее информативен, и из данного фрагмента можно заключить только, что для локальных переменных отведено 84 байта. Саму же структуру локальных переменных можно понять только путем анализа кода, который следует ниже.

При обращении к коду, сгенерированному транслятором Visual C++, мы выясняем, что распознать там локальные переменные не всегда возможно. Транслятор обладает мощным оптимизатором кода и в некоторых случаях просто опускает локальные переменные, часто обходясь без использования

регистра ESP. Переменная превращается либо в константу, либо хранится в регистре, что, несомненно, оптимизирует полученный код. Например, рассмотрим следующий фрагмент, из некоторой функции:

```
...
int b=10;
...
c=c+b;
```

Переменная b — локальна. Если, кроме указанного действия, она нигде не используется — следовательно, она, по сути, является константой. Поэтому транслятор не резервирует для нее стековую память, а просто записывает команду типа ADD EAX,10, предполагая, что переменная c хранится в регистре EAX. В следующем разделе будет приведен пример подобного поведения транслятора Visual C++.

Функции и процедуры

Функции и процедуры идентифицируются достаточно просто. Структуру вызова и внутреннюю часть процедур вы уже хорошо знаете. Остается только напомнить некоторые положения.

Вызов процедуры:

```
PUSH par1
PUSH par2
PUSH par3
CALL 232343
```

Здесь все достаточно просто. Главное — распознать параметры и понять порядок помещения их в стек. Надо также иметь в виду, что существует протокол передачи параметров через регистры *(см. главу 3.7)*. Например, при быстром вызове fastcall (для компиляторов от Borland) первые три параметра помещаются в регистры EAX, ECX, EDX, а остальные, если они есть, помещаются в стек обычным способом. После вызова процедуры может стоять команда очистки стека ADD ESP,N.

Внутренняя часть процедуры также нами неоднократно разбиралась *(см. главы 1.2 и 3.7)*. Думаю, что она достаточно вами узнаваема, и мы не будем здесь подробно на этом останавливаться. Имейте только в виду, что наличие локальных переменных не означает автоматически, что для них будет выделяться место в стеке и предоставляться для использования регистр EBP. В целях оптимизации транслятор может заменить переменную константным выражением либо использовать для ее хранения регистр общего назначения. Для доступа же к параметрам будет использоваться регистр ESP. Чтобы не быть голословным, рассмотрим простую функцию на языке C:

```
int func(int a, int b)
{
  int i;
  i=a+b+0x1234;
  return i;
}
```

После трансляции Visual C++ мы приходим к достаточно удивительному результату:

```
:00401000 8B442408          mov eax, dword ptr [esp+08]
:00401004 8B4C2404          mov ecx, dword ptr [esp+04]
:00401008 8D840134120000    lea eax, dword ptr [ecx+eax+00001234]
:0040100F C3                ret
```

Как видим, нет резервирования локальной переменной и не используется регистр `EBP`. Но ведь, по сути, все правильно. Зачем резервировать память для переменной, если можно обойтись регистрами?

Не забывайте, наконец, что функции возвращают результат через регистр `EAX` либо через пару `EDX:EAX` для 64-битной переменной. Это может помочь вам быстро разобраться в назначении функции.

Оптимизация кода

Выше мы пытались восстановить исходный текст программы по ассемблерному коду. Насколько это удалось, можно выяснить, сравнив получившийся текст с исходным. Следует заметить, что мы ошиблись: в исходном тексте стоит цикл `while`, а в нашем — `do`. Однако если откомпилировать получившуюся программу, она будет работать так же, как и исходная. Причина отличия двух текстов программ (весьма простых, кстати) заключается в том, что транслятор, кроме всего прочего, производит еще оптимизацию кода. Результат оптимизации — принципиальная невозможность восстановить исходный текст по исполняемому коду. Можно, однако, получить программный текст, правильно описывающий алгоритм исполнения, или просто понять, что делает данный код, чем, собственно, мы и занимаемся в данной главе.

Рассмотрим небольшую и весьма тривиальную программу на С (листинг 4.4.5).

Листинг 4.4.5. Пример небольшой программы на С

```
void main ()
{
  int i;
  char a[10];
  char b[11];
  for(i=0; i<10; i++)
```

```
  {
    a[i]='a';
    *(b+i)='a';
    printf("%c %c\n",a[i],b[i]);
  }
  ExitProcess(0);
}
```

А вот как выглядит ассемблерный код программы из листинга 4.4.5, полученный с помощью транслятора Borland C++ 5.0 (листинг 4.4.6).

Листинг 4.4.6. Дизассемблированный текст программы (см. листинг 4.4.5). Транслятор Borland C++ 5.0

```
.text:00401108 _main proc near ; DATA XREF: .data:0040A0B8
.text:00401108
.text:00401108 var_18 = byte ptr -18h
.text:00401108 var_C  = byte ptr -0Ch
.text:00401108 argc   = dword ptr 8
.text:00401108 argv   = dword ptr 0Ch
.text:00401108 envp   = dword ptr 10h
.text:00401108
.text:00401108        push  ebp
.text:00401109        mov   ebp, esp
.text:0040110B        add   esp, 0FFFFFFE8h
.text:0040110E        push  ebx
.text:0040110F        xor   ebx, ebx
.text:00401111
.text:00401111 loc_401111: ; CODE XREF: _main+30
.text:00401111        mov  [ebp+ebx+var_C], 61h
.text:00401116        mov  [ebp+ebx+var_18], 61h
.text:0040111B        movsx eax, [ebp+ebx+var_18]
.text:00401120        push  eax
.text:00401121        movsx edx, [ebp+ebx+var_C]
.text:00401126        push  edx        ; char
.text:00401127        push  offset aCC ; __va_args
.text:0040112C        call  _printf
.text:00401131        add   esp, 0Ch
.text:00401134        inc   ebx
.text:00401135        cmp   ebx, 0Ah
.text:00401138        jl    short loc_401111
.text:0040113A        push  0 ; uExitCode
.text:0040113C        call  ExitProcess
.text:00401141        pop   ebx
.text:00401142        mov   esp, ebp
```

```
.text:00401144         pop   ebp
.text:00401145         retn
.text:00401145 _main   endp
```

Кстати, функция `ExitProcess(0)` введена в текст программы для быстрого поиска нужного фрагмента в отладчике или дизассемблированном коде. Невооруженным глазом видно, что оптимизацией здесь и не пахло. По такому тексту достаточно просто восстановить исходную C-программу. А вот код, оптимизированный транслятором Visual C++ (листинг 4.4.7).

Листинг 4.4.7. Дизассемблированный текст программы (см. листинг 4.4.5). Транслятор Visual C++

```
.text:00401000 sub_401000 proc near ; CODE XREF: start+AF
.text:00401000         push  esi
.text:00401001         mov   esi, 0Ah
.text:00401006
.text:00401006 loc_401006: ; CODE XREF: sub_401000+18
.text:00401006         push  61h
.text:00401008         push  61h
.text:0040100A         push  offset aCC ; "%c %c\n"
.text:0040100F         call  sub_401030 ;printf
.text:00401014         add   esp, 0Ch
.text:00401017         dec   esi
.text:00401018         jnz   short loc_401006
.text:0040101A         push  0  ; uExitCode
.text:0040101C         call  ds:ExitProcess
.text:00401022         pop   esi
.text:00401023         retn
.text:00401023 sub_401000   endp
```

Я думаю, что текст из листинга 4.4.7 удивит вас. Однако что же здесь удивительного? Взгляните на текст C-программы. Заданные нами два символьных массива — абсолютно ни к чему. Транслятор Visual C++ очень точно это подметил и изменил код так, что, как ни старайся, текст исходной программы восстановить не удастся. Конечно, такая оптимизация оказалась возможной только потому, что наши массивы используются в ограниченной области.

Рассмотрим далее некоторые способы оптимизации, которые могут пригодиться нам и при написании программ на ассемблере.

- *Скорость или объем.* Это весьма важный вопрос, когда дело касается микропроцессора. Например, команда `MOV EAX,EBX` выполняется быстрее команды `XCHG EAX,EBX`, но код ее длиннее. Зная такую особенность, можно либо сокращать объем, либо увеличивать скорость программы. Особенно

часто используется замена такой операции, как `MUL`, другими командами, в частности `SHL`, а операции `DIV` на `SHR`. Это может значительно увеличить скорость выполнения программы и увеличить ее объем. Интересно, что арифметические действия можно производить и с помощью команды `LEA` *(см. приложение 2)*, и современные трансляторы уже взяли это на вооружение. Так что команда `MUL` не так часто может встретиться в оттранслированном коде, как об этом можно подумать, исходя из текста программ. Вообще, свойство команд надо исследовать, и иногда узнаешь довольно интересные вещи. Например, нахождение остатка от деления числа без знака на 4 производится следующим образом: `AND  EAX,0003h` — не правда ли, оригинально?

- *Оптимизация условных переходов.* Оказывается, здесь тоже имеются резервы. Можно построить проверку условия так, чтобы количество переходов было наименьшим. Таким образом, можно добиться того, чтобы данный фрагмент программы работал несколько быстрее. Предположим, у вас имеется следующий фрагмент.

```
    ...
    CMP EAX,100
    JB  L1

;Фрагмент 1
    ...
    JMP L2
L1:

;Фрагмент 2
    ...
L2:
```

Мы знаем, что содержимое `EAX` чаще всего оказывается меньше 100. Следовательно, фрагмент можно заменить на следующий:

```
    ...
    CMP EAX,100
    JNB L1

;Фрагмент 2
    ...
    JMP L2
L1:

;Фрагмент 1
    ...
L2:
```

- *Оптимизация вызовов процедур.* Рассмотрим следующий фрагмент:

```
P1 PROC
    ...
    CALL P2
    RET
P1 ENDP
...
P2 PROC
    ...
    RET
P2 ENDP
```

Данный фрагмент можно заменить более эффективным. Подобный подход можно часто встретить, просматривая отладчиком код программы.

```
P1 PROC
    ...
    JMP P2
P1 ENDP
...
P2 PROC
  ...
  RET
P2 ENDP
```

Код становится и быстрее, и короче, вот только разобраться в нем становится сложнее. На этом мы оставляем вопрос оптимизации. Всех интересующихся могу отослать к книгам [14, 27].

Объектное программирование

Объектное программирование в значительной степени усложняет получаемый исполняемый код. Мы рассмотрим один простой пример. В листинге 4.4.8 представлена программа на C++.

Листинг 4.4.8. Программа на C++, использующая объекты

```
#include <windows.h>
#include <stdio.h>
class stroka {
      public:
      char c[200];
      char g[100];
      stroka() {strcpy(c,"Privet");strcpy(g,"Privet");}
      int strcp();
};
```

```
 stroka::stri()
{
  strcpy(c,g);
  return 0;
}
main()
{
  stroka * s = new stroka;
  s->strcp();
  printf("%s\n",s->c);
  delete s;
}
```

В листинге 4.4.9 представлен дизассемблированный код функции `main`.

Листинг 4.4.9. Дизассемблированный код функции `main` из листинга 4.4.8. Дизассемблер IDA Pro

```
CODE:00401122 _main proc near ; DATA XREF: DATA:0040C044
CODE:00401122
CODE:00401122 var_28 = dword ptr -28h
CODE:00401122 var_18 = word ptr -18h
CODE:00401122 dest   = dword ptr -4
CODE:00401122 argc   = dword ptr 8
CODE:00401122 argv   = dword ptr 0Ch
CODE:00401122 envp   = dword ptr 10h
CODE:00401122
CODE:00401122        push  ebp
CODE:00401123        mov   ebp, esp
CODE:00401125        add   esp, 0FFFFFFD8h
CODE:00401128        push  ebx
CODE:00401129        mov   eax, offset stru_40C084
CODE:0040112E        call  @__InitExceptBlockLDTC
CODE:00401133        push  12Ch
CODE:00401138        call  unknown_libname_8
CODE:0040113D        pop   ecx
CODE:0040113E        mov  [ebp+dest], eax
CODE:00401141        test  eax, eax
CODE:00401143        jz    short loc_40117D
CODE:00401145        mov  [ebp+var_18], 14h
CODE:0040114B        push  offset aPrivet   ; src
CODE:00401150        push  [ebp+dest]       ; dest
CODE:00401153        call  _strcpy
CODE:00401158        add   esp, 8
CODE:0040115B        push  offset aPrivet_0 ; src
CODE:00401160        mov   edx, [ebp+dest]
```

```
CODE:00401163         add   edx, 0C8h
CODE:00401169         push  edx              ; dest
CODE:0040116A         call  _strcpy
CODE:0040116F         add   esp, 8
CODE:00401172         mov  [ebp+var_18], 8
CODE:00401178         mov   ebx, [ebp+dest]
CODE:0040117B         jmp   short loc_401180
CODE:0040117D ; -------------------------------------
CODE:0040117D
CODE:0040117D loc_40117D:   ; CODE XREF: _main+21
CODE:0040117D         mov   ebx, [ebp+dest]
CODE:00401180
CODE:00401180 loc_401180:  ; CODE XREF: _main+59
CODE:00401180         push  ebx         ; dest
CODE:00401181         call  sub_401108
CODE:00401186         pop   ecx
CODE:00401187         push  ebx         ; char
CODE:00401188         push  offset aS   ; __va_args
CODE:0040118D         call  _printf
CODE:00401192         add   esp, 8
CODE:00401195         push  ebx         ; block
CODE:00401196         call @$bdele$qpv ; operator delete(void *)
CODE:0040119B         pop   ecx
CODE:0040119C         mov   eax, [ebp+var_28]
CODE:0040119F         mov   large fs:0, eax
CODE:004011A5         xor   eax, eax
CODE:004011A7         pop   ebx
CODE:004011A8         mov   esp, ebp
CODE:004011AA         pop   ebp
CODE:004011AB         retn
CODE:004011AB _main   endp
```

Как видите, получившийся код достаточно сложен. Я не намерен проводить его детальный анализ, т. к. в этом случае нам пришлось бы включать в него и анализ библиотечных программ. Остановимся только на некоторых ключевых моментах.

- Оператор `NEW` сводится к выполнению библиотечной процедуры: `unknown_libname_8`. Последняя процедура выделяет память для свойств экземпляра объекта (300 байтов).
- Конструктор хранится и выполняется непосредственно в теле программы. Это связано с тем, что и сам конструктор определен в тексте класса. Для эксперимента попробуйте вынести текст конструктора в отдельную функцию, и вы увидите, что конструктор будет вызываться из `main`, как обычная функция.

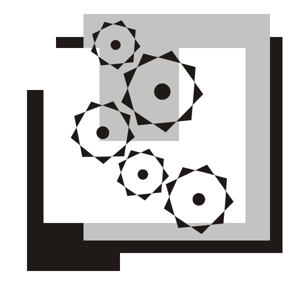
- Процедуру `@__InitExceptBlockLDTC` вставляет транслятор для поддержки обработки исключений (exception). Вы можете удалить информацию, необходимую для использования исключений, несколько сократив исполняемый код, но тогда вы не сможете использовать операторы исключений, такие как `try` или `catch`.
- При вызове какого-либо метода в стек всегда помещается, по крайней мере, один параметр — указатель на экземпляр объекта.

Приведу теперь фрагмент той же дизассемблированной программы, но с использованием опции `-x`, что для транслятора Borland C++ (см. листинг 4.4.10) означает: не поддерживать обработку исключений. Как видите, текст программы оказался значительно проще.

Листинг 4.4.10. Дизассемблированный код функции `main` из листинга 4.4.7, при трансляции использована опция `-x` (исключить исключения, Borland C++). Дизассемблер IDA Pro

```
.text:00401122 _main proc near ; DATA XREF: .data:0040B0C8
.text:00401122
.text:00401122 argc   = dword ptr 8
.text:00401122 argv   = dword ptr 0Ch
.text:00401122 envp   = dword ptr 10h
.text:00401122
.text:00401122        push  ebp
.text:00401123        mov   ebp, esp
.text:00401125        push  ebx
.text:00401126        push  12Ch
.text:0040112B        call  unknown_libname_8
.text:00401130        pop   ecx
.text:00401131        mov   ebx, eax
.text:00401133        test  eax, eax
.text:00401135        jz    short loc_40115D
.text:00401137        push  offset aPrivet   ; src
.text:0040113C        push  ebx              ; dest
.text:0040113D        call  _strcpy
.text:00401142        add   esp, 8
.text:00401145        push  offset aPrivet_0 ; src
.text:0040114A        lea   edx, [ebx+0C8h]
.text:00401150        push  edx              ; dest
.text:00401151        call  _strcpy
.text:00401156        add   esp, 8
.text:00401159        mov   ecx, ebx
.text:0040115B        jmp   short loc_40115F
.text:0040115D ;---------------------------------------------
.text:0040115D
```

```
.text:0040115D loc_40115D: ; CODE XREF: _main+13
.text:0040115D        mov   ecx, ebx
.text:0040115F
.text:0040115F loc_40115F: ; CODE XREF: _main+39
.text:0040115F        mov   ebx, ecx
.text:00401161        push  ebx        ; dest
.text:00401162        call  sub_401108
.text:00401167        pop   ecx
.text:00401168        push  ebx        ; char
.text:00401169        push  offset aS  ; __va_args
.text:0040116E        call  _printf
.text:00401173        add   esp, 8
.text:00401176        push  ebx        ; handle
.text:00401177        call  __rtl_close
.text:0040117C        pop   ecx
.text:0040117D        xor   eax, eax
.text:0040117F        pop   ebx
.text:00401180        pop   ebp
.text:00401181        retn
.text:00401181 _main  endp
```



Глава 4.5

Исправление исполняемых модулей

В данной главе будут представлены два примера исследования и исправления исполняемого модуля. Хочу подчеркнуть, что данный материал приведен здесь только в качестве учебных целей и не может быть использован в противоправных действиях. Исправлять исполняемые модули приходится и на вполне законных основаниях. Кроме того, разработчики программного обеспечения должны быть знакомы с методами взлома, дабы строить защиту своих программ более профессионально.

Простой пример исправления исполняемого модуля

Сейчас мы рассмотрим простой[1] пример, демонстрирующий некоторые приемы такого типа работы. Задача, которую ставим перед собой, не так сложна, и решить ее можно, воспользовавшись только дизассемблером W32Dasm.

Данная программа (Allscreen — программа, с помощью которой можно "снимать" окна и отдельные части экрана) попала ко мне как Shareware Release. Программа написана на Delphi, но мы увидим, что решить поставленную задачу можно, и не зная, на чем написана программа. При запуске ее на экране появляется окно, изображенное на рис. 4.5.1. Ближе познакомившись с предметом, вы убедитесь, что чаще всего приходится искать место в программе, соответствующее какому-либо визуальному эффекту: открытие окна, закрытие окна, вывод текста и т. п.

---

[1] Простой с точки зрения возможных проблем, возникающих при исправлении исполняемых модулей.



Рис. 4.5.1. Окно, появляющееся при запуске программы Allscreen



Рис. 4.5.2. Окно задержки

При нажатии кнопки **Accept** возникает задержка секунд в шесть (рис. 4.5.2). Далее программа работает нормально.

После 15-ти запусков появляется окно, представленное на рис. 4.5.3, и происходит выход из программы.



Рис. 4.5.3. Сообщение об истечении времени работы программы

Таким образом, следует решить две задачи:

- устранить весьма раздражающую задержку;
- сделать так, чтобы программа работала при любом количестве запусков.

Окно на рис. 4.5.2 являет собой явный "прокол" авторов программы. Дело в том, что окно и все его содержимое можно спрятать в ресурсы. Но когда на том же окне появляется новая запись — это уже программный код. Итак, запускаем W32Dasm и считываем туда программу Allscreen. Запускаем окно SDR (String Data Refrence), ищем строку Shareware Delay, дважды щелкаем по ней и, закрыв его, оказываемся в нужном месте программы. Вот этот фрагмент (листинг 4.5.1).

Листинг 4.5.1. Фрагмент кода, осуществляющего, в частности, задержку

```
* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004420BC(C)
|
:00442123 33D2                xor  edx, edx
:00442125 8B83B0010000        mov  eax, dword ptr [ebx+000001B0]
:0044212B E8541DFDFF          call 00413E84
:00442130 33D2                xor  edx, edx
:00442132 8B83B4010000        mov  eax, dword ptr [ebx+000001B4]
:00442138 E8471DFDFF          call 00413E84
:0044213D 33D2                xor  edx, edx
:0044213F 8B83B8010000        mov  eax, dword ptr [ebx+000001B8]
:00442145 E83A1DFDFF          call 00413E84
:0044214A BA50000000          mov  edx, 00000050
:0044214F 8B83BC010000        mov  eax, dword ptr [ebx+000001BC]
:00442155 E8D618FDFF          call 00413A30

* Possible StringData Ref from Code Obj ->"Shareware Delay"
                 |
:0044215A BAA8214400          mov  edx, 004421A8
:0044215F 8B83BC010000        mov  eax, dword ptr [ebx+000001BC]
:00442165 E8EE1DFDFF          call 00413F58
:0044216A 33D2                xor  edx, edx
:0044216C 8B83C0010000        mov  eax, dword ptr [ebx+000001C0]
:00442172 E80D1DFDFF          call 00413E84
:00442177 33D2                xor  edx, edx
:00442179 8B83C4010000        mov  eax, dword ptr [ebx+000001C4]
:0044217F E8001DFDFF          call 00413E84
:00442184 33D2                xor  edx, edx
:00442186 8B83C8010000        mov  eax, dword ptr [ebx+000001C8]
:0044218C E8F31CFDFF          call 00413E84
:00442191 8B83CC010000        mov  eax, dword ptr [ebx+000001CC]
:00442197 E8E8D4FFFF          call 0043F684
:0044219C 5B                  pop  ebx
:0044219D C3                  ret
```

Я сразу взял несколько больше кода, захватив и несколько верхних строк. По сути дела, перед нами вся процедура задержки. Нет смысла пытаться понять, что означает та или иная команда `call`, хотя легко сообразить (проведя небольшой эксперимент), что, например, `call 00413E84` убирает строку с экрана.

Для того чтобы решить проблему задержки, достаточно "выключить" этот фрагмент из программы. Проще всего это можно сделать, поставив в начало

команды `pop ebx / ret`, используя такой редактор, как Hiew. В результате задержка исчезает.

Перейдем теперь ко второй проблеме — ограничение на количество запусков (листинг 4.5.2). Уже из самого вида окна ясно, что оно формируется в самой программе. Следовательно, опять можно попытаться найти текст, который изображается на экране, в самой программе.

Листинг 4.5.2. Фрагмент кода проверки количества запусков

```
:00443326 8BC0                   mov  eax, eax
:00443328 53                     push ebx
:00443329 8BD8                   mov  ebx, eax
:0044332B 803DEC56440001         cmp  byte ptr [004456EC], 01
:00443332 7546                   jne  0044337A
:00443334 A124564400             mov  eax, dword ptr [00445624]
:00443339 E84E2CFEFF             call 00425F8C
:0044333E A1D8564400             mov  eax, dword ptr [004456D8]
:00443343 E87816FEFF             call 004249C0
:00443348 FF05F0564400           inc  dword ptr [004456F0]
:0044334E C605EC56440000         mov  byte ptr [004456EC], 00
:00443355 833DF05644000F         cmp  dword ptr [004456F0], 0000000F
:0044335C 7E1C                   jle  0044337A
:0044335E 6A00                   push 00000000
:00443360 668B0DB0334400         mov  cx, word ptr [004433B0]
:00443367 B202                   mov  dl, 02

* Possible StringData Ref from Code Obj ->"This Software Has Been Used Over"
:00443369 B8BC334400             mov  eax, 004433BC
:0044336E E8BDAEFEFF             call 0042E230
:00443373 8BC3                   mov  eax, ebx
:00443375 E84214FEFF             call 004247BC

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:00443332(C), :0044335C(C)
|
:0044337A 33D2                   xor  edx, edx
:0044337C 8B83F4010000           mov  eax, dword ptr [ebx+000001F4]
:00443382 E8A52DFFFF             call 0043612C
:00443387 33D2                   xor  edx, edx
:00443389 8B83F8010000           mov  eax, dword ptr [ebx+000001F8]
:0044338F E8982DFFFF             call 0043612C
:00443394 33D2                   xor  edx, edx
:00443396 8B83FC010000           mov  eax, dword ptr [ebx+000001FC]
:0044339C E88B2DFFFF             call 0043612C
```

```
:004433A1 33D2                  xor  edx, edx
:004433A3 8B8314020000          mov  eax, dword ptr [ebx+00000214]
:004433A9 E87E2DFFFF            call 0043612C
:004433AE 5B                    pop  ebx
:004433AF C3                    ret
```

Опять мы представляем весь необходимый фрагмент. Используя найденную выше ссылку на искомую строку, легко обнаруживаем "подозрительные" команды:

```
cmp dword ptr [004456F0], 0000000F
jle 0044337A
```

Вспомним, что программа перестает работать как раз после пятнадцати запусков. Проще всего исправить ситуацию, "забив" фрагмент программы с `0044335E` по `00443375` командами `NOP (90)`, используя редактор Hiew.

Пример снятия защиты

Перед вами пример того, как часто до цели добираешься длинным окружным путем, вместо того, чтобы быстро пройти по короткой и ровной дороге. На этот раз объектом исследования станет программа GetPixel.exe. Программа предназначена для "снятия" с экрана цветовых пикселов. Она попала ко мне уже вместе с программой crack.exe. Для тех, кто не знает: так обычно называют программы для снятия защиты. А поскольку мне необходим был учебный пример, я предпочел снимать защиту без посторонней помощи (и интересно, и полезно). Замечу, что программа написана на языке Visual Basic, но я никак не использую это знание для исследования кода. Во всяком случае, известные мне декомпиляторы языка Visual Basic не дали сколько-нибудь полезных результатов.

Стадия 1. Попытка зарегистрироваться

На рис. 4.5.4 представлено окно программы GetPixel.exe, которое по замыслу автора должно использоваться для регистрации пользователя. Поля **Name** и **Registration Code**[2] предназначены, соответственно, для ввода имени регистрируемого и кода регистрации. При нажатии кнопки **OK** делается проверка имени и пароля. Разумеется, когда я ввел произвольное имя и пароль, программа выдала сообщение, что я ошибся. Иного я и не ожидал.

[2] Непрописанное на рисунке слово "Code" — не моя вина: так работает программа.

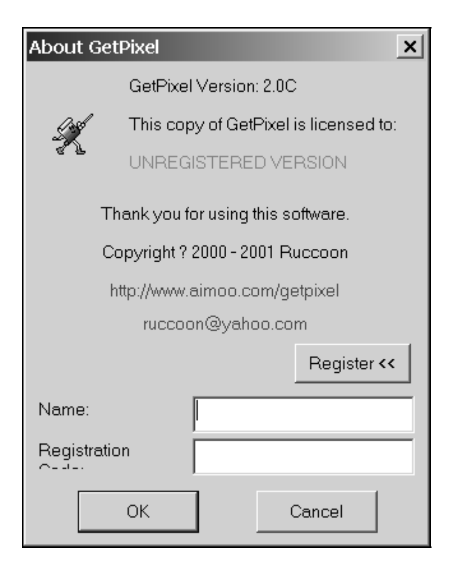


Рис. 4.5.4. Окно регистрации программы GetPixel

Ну что же, попробуем заставить программу зарегистрировать мое имя и пароль. По логике предпринимаемых мною действий, начнем с поиска строк, содержащих слово "Registration".

Открываем знаменитый дизассемблер IDA Pro и загружаем туда нашу программу. В окне **Strings** находим сразу три строки, содержащих данное слово: "Register Successfully!", "Registration", "Register Fail!". Ага, похоже, мы на верном пути. Начнем с первой фразы. Щелкнув дважды по строке, оказываемся в нужном месте окна дизассемблера:

```
.text:00409720        aRegisterSucces:  ; DATA XREF: .text:00417ECE↓o
.text:00409720        unicode 0, <Register Successfully!>,0
```

И далее по ссылке находим следующий фрагмент:

```
.text:00417EC5        lea  edx, [ebp-134h]
.text:00417ECB        lea  ecx, [ebp-34h]
.text:00417ECE        mov  dword ptr [ebp-12Ch], offset aRegisterSucces ; "Register
Successfully!"
.text:00417ED8        mov  dword ptr [ebp-134h], 8
.text:00417EE2        call ds:__vbaVarDup
```

Что собой представляет функция `__vbaVarDup`, сказать трудно, но похоже на сообщение — сообщение об удачной регистрации. Попробуем подробнее изучить текст программы вблизи данных строк. Чуть выше по тексту фрагмента обнаруживаем:

```
.text:00417E76        push ecx
.text:00417E77        push edx
```

```
.text:00417E78         push 4
.text:00417E7A         call edi ; __vbaFreeVarList
.text:00417E7C         add  esp, 20h
.text:00417E7F         cmp  [ebp-1A8h], bx
.text:00417E86         jz   loc_4181A4
```

Это уже настораживает. Посмотрим, что находится по адресу `loc_4181A4`. Переходим и чуть ниже обнаруживаем еще один фрагмент:

```
.text:00418268         mov  dword ptr [ebp-12Ch], offset aRegisterFailed ; "Register
Failed!"
.text:00418272         mov  dword ptr [ebp-13Ch], offset aPleaseVisit
; "Please visit"
.text:0041827C         mov  dword ptr [ebp-14Ch], offset aHttpWww_aimoo_ ;
"http://www.aimoo.com/getpixel"
.text:00418286         mov  dword ptr [ebp-15Ch], offset aToGetYourRegis ; "to get
your register code"
.text:00418290         call ebx ; __vbaVarCat
```

Ага, сообщение о неудачной регистрации и приглашение на сайт. Да, похоже, мы на верном пути. Запускаем hiew32.exe и находим адрес `.text:00417E86`. Вводим 6 байтов `90h` (`NOP`). Выходим, запускаем программу. Далее в окне регистрации вводим произвольное имя и код и получаем сообщение, что зарегистрированы. Ура, задача решена!? Однако не тут-то было.

Стадия 2. Избавляемся от надоедливого окна

Но трудности этим не заканчиваются. Пока после регистрации я не выходил из программы, окно регистрации сообщало, что мы зарегистрировались. После перезапуска окно опять стало показывать, что копия не зарегистрирована. Кроме этого, при перезапусках с некоторой вероятностью стало появляться окно, представленное на рис. 4.5.5. При нажатии кнопки **Да** программа пытается выйти на сайт создателя, в случае нажатия кнопки **Нет** программа продолжает свою работу обычным способом.

В том же каталоге, где расположена программа, я обнаруживаю файл cklickme.reg, который содержит скрипт[3] для записи в реестр правильного имени и пароля, если, конечно, мы их знаем. Обращаюсь к реестру по найденному в скрипте адресу. Оказалось, что те имя и пароль, которые мы ввели, записались именно туда. По-видимому, программа при запуске сравнивает их с некоторыми эталонными значениями, которые мы не знаем и, забегая впе-

[3] Термин "скрипт" (от англ. *script* — сценарий) уже вполне прижился в литературе по программированию. Так что будем употреблять его, вместо "сценарий", имеющего в русском языке слишком большой диапазон значений.

ред, не узнаем, а далее указывает в окне, что программа так и не зарегистрирована. Кроме этого с некоторой вероятностью при запуске выдается окно-"надоедало" (рис. 4.5.5).



Рис. 4.5.5. Окно-"надоедало"

Но давайте действовать по порядку и разберемся сначала с надоедливым окном. Начнем с поиска строки "How do you feel me?" в окне **Strings** IDA Pro. Легко находим ее и обращаемся к участку кода, где есть ссылка на эту строку. Вот эти строки:

```
.text:0040B217      push eax
.text:0040B218      mov  dword ptr [ebp-0D0h], offset aHowDoYouFeelMe ; "How do you
feel me?"
.text:0040B222      mov  dword ptr [ebp-0E0h], offset aIWantToBeConfi ; "I want to
be confirmed :-)"
.text:0040B22C      call esi ; __vbaVarCat
.text:0040B22E      lea  ecx, [ebp-0E8h]
.text:0040B234      push eax
.text:0040B235      lea  edx, [ebp-98h]
.text:0040B23B      push ecx
.text:0040B23C      push edx
.text:0040B23D      call esi ; __vbaVarCat
.text:0040B23F      push eax
.text:0040B240      call ds:rtcMsgBox
```

Очевидно, что `call rtcMsgBox` — это как раз вызов функции `MessageBox`. Разумеется, эта функция нам не нужна и мы можем забить ее `NOP`. Но погодите. Ведь она должна предлагать выбор, и мы должны выбрать **Нет**. Опустимся немного вниз. Вот этот фрагмент:

```
.text:0040B2B6      call ds:__vbaVarTstEq
.text:0040B2BC      test ax, ax
.text:0040B2BF      jz   short loc_40B305
.text:0040B2C1      mov  esi, ds:__vbaStrToAnsi
.text:0040B2C7      push 1
.text:0040B2C9      lea  edx, [ebp-60h]
.text:0040B2CC      push offset aC       ; "C:\\"
.text:0040B2D1      push edx
```

```
.text:0040B2D2      call esi ; __vbaStrToAnsi
.text:0040B2D4      push eax
.text:0040B2D5      push 0
.text:0040B2D7      lea  eax, [ebp-5Ch]
.text:0040B2DA      push offset aHttpWww_aimoo_ ; "http://www.aimoo.com/getpixel"
.text:0040B2DF      push eax
.text:0040B2E0      call esi ; __vbaStrToAnsi
.text:0040B2E2      push eax
.text:0040B2E3      push 0
.text:0040B2E5      push 0
.text:0040B2E7      call sub_407CB0
.text:0040B2EC      call ds:__vbaSetSystemError
```

Очевидно, что если условие равенства нулю содержимого регистра `EAX` не выполнится, то как раз и будет вызываться сайт автора. Таким образом, следует заменить `JZ` на `JMP SHORT` — и все.

Запускаем hiew32.exe и вносим изменения в два указанных фрагмента. Не забываем, что забивать надо и сам вызов процедуры, и команды `PUSH` к ней. В результате действительно раздражающее меня окно перестает запускаться.

Стадия 3. Доводим регистрацию до логического конца

Ну что же, теперь осталось последнее — заставить программу поверить, что в реестре находятся правильные регистрационные данные. По-видимому, разумно предположить, что есть какая-то процедура, которая и проверяет правильность пароля.

Признаюсь, что тут я плутал около часа, используя попеременно то дизассемблер, то отладчик OllyDbg. Мне надо было сразу сообразить, как добраться до этой процедуры. Вот этот путь, я вам сейчас и опишу.

Прежде всего, надо было обратить внимание на название полей в реестре, которые заполняются при регистрации. Это поля **License** и **RegUser**. В поле **License** как раз пароль и записывается. Вот и поищем эту строку.

Строку мы обнаруживаем. Она встречается в двух местах. Это уже обнадеживает: к паролю обращаются при запуске программы и из окна регистрации. Глядим на дизассемблированный текст и видим, что в одном случае используется функция `rtcGetSetting`, а во втором — функция `rtcSaveSetting`. Ну, тут уже все ясно. Первая функция читает пароль, а вторая записывает. Данные об этих функциях есть в MSDN. Очевидно, что нам следует обратить внимание именно на первую функцию.

Переходим к нужному фрагменту и, спускаясь по тексту вниз, пытаемся понять логику программы. Двигаясь вниз, будем отслеживать только не библиотечные процедуры. Какая-то из них, скорее всего, и будет процедурой проверки правильности пароля и имени.

Вначале мое внимание привлек следующий фрагмент:

```
.text:0040AE00        lea  edx, [ebp-78h]
.text:0040AE03        push edx
.text:0040AE04        call sub_415160
```

Что помещают в `EDX`? Запускаем отладчик, ставим точку прерывания на адрес `40ae03`. Смотрим, на что указывает в стеке регистр `EDX`. Оказалось, что на имя, полученное из реестра. Пароля здесь нет. Следовательно, процедура не та, что нам нужна. И мы двигаемся далее. А вот это уже интересно:

```
.text:0040AE5C        lea  edx, [ebp-88h]
.text:0040AE62        lea  eax, [ebp-78h]
.text:0040AE65        push edx
.text:0040AE66        push eax
.text:0040AE67        call sub_416070
```

Из отладчика узнаем, что `EDX` указывает на строку, состоящую из имени и пароля — где-то по дороге их объединили. Выполняем в отладчике процедуру — она возвращает `0` в регистре `EAX`, а `0` во многих языках — это `false`. Ну что же, пожалуй, пришло время эксперимента.

Запускаем hiew32.exe и вместо фрагмента

```
.text:0040AE65        push edx
.text:0040AE66        push eax
.text:0040AE67        call sub_416070
```

ставим команду `MOV EAX,1`, а остальные байты забиваем байтами `90H`. Запускаем программу, входим в окно регистрации... И, о радость, мы зарегистрированы!

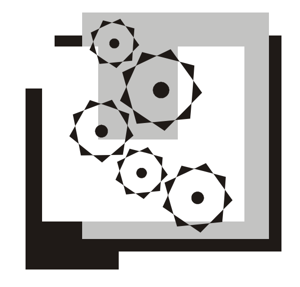
Стадия 4. Неожиданная развязка

Теперь я вам скажу, что имеется куда более короткая дорога к правильному результату. Ну конечно, наверное, вы уже догадались. Нужно просто обратиться по адресу `00416070H`, т. е. адресу, где начинается процедура проверки пароля, и в самом начале ее поставить всего две команды: `MOV EAX,1`, `RETN 8`. И все, больше ничего не надо. Не нужны все три, описанные здесь стадии. Я мог бы, конечно, сразу дать читателю готовое короткое решение. Но:

- при реальном анализе часто задача решается как раз не самым коротким путем. Короткое изящное решение приходит после. А, следовательно, обучаться на красивых решениях — это неправильный прием;

- в конце концов, какая разница, каким путем шел исследователь, важно другое — задача решена.

Я думаю, у читателя возник еще один, более частный вопрос. В своем решении я основывался на информации, полученной из найденного в каталоге скрипта. Из него я узнал, где записываются имя и пароль при регистрации программы. А если бы не было скрипта? Да нет проблем! Можно воспользоваться каким-нибудь монитором, отслеживающим доступ к реестру. А если нет монитора, то и прямой анализ дизассемблируемого текста вполне годится. Функции `rtcSaveSetting` и `rtcGetSetting` так и напрашиваются для анализа.



Глава 4.6

Структура и написание драйверов

Материал, который теперь мы намерены разобрать, относится к категории сложного[1]. Сразу хочу предупредить, что приступать к нему следует, разобравшись, во-первых, в страничной адресации *(см. главу 3.6)*, во-вторых, в программировании и управлении сервисами *(см. главу 3.8)*. Я в своем изложении буду опираться на эти материалы. Кроме этого в *приложении 3* имеется краткий обзор по защищенному режиму процессоров Intel, следует ознакомиться и с этой информацией.

Драйверы режима ядра, разбираемые в данной главе, могут работать только в операционной системе Windows семейства NT.

О ядре и структуре памяти

Прежде рассмотрим, что же такое ядро операционной системы. Ядро — это часть операционной системы, хранящаяся в оперативной памяти. Ясно, что ядро должно быть как-то защищено от попыток доступа (злонамеренных или случайных) прикладных программ. Совершенно очевидно, что полностью защитить ядро невозможно без какой-либо аппаратной поддержки. Такой аппаратной поддержкой в частности является защищенный режим процессора Intel. В *главе 3.6* мы уже говорили о защищенном режиме применительно к управлению памятью. Кроме этого в *приложении 3* кратко излагаются основы функционирования защищенного режима. И я надеюсь, что читатель имеет достаточное представление об этом. Посмотрите на рис. 3.6.4. Там изображено адресное пространство процесса. Использованием страничного механизма достигается удивительный эффект — адресное пространство

---

[1] Программные модули, представленные далее, были протестированы в Windows XP, Windows Server 2003, Windows Vista.

намного превышает реальный объем физической памяти. При этом реальная страница может храниться не в памяти, а в так называемом страничном файле. Это интересно, но я не намерен обсуждать это в данной книге.

Мне бы хотелось обсудить вот какой вопрос. Дело в том, что в процессоре Intel одновременно работают два механизма формирования адресов памяти. Это сегментный и страничный механизмы. Причем страничный механизм функционирует на более низком уровне. С другой стороны, если говорить о защите, то защита на уровне сегментов обладает большим приоритетом, чем на уровне страниц. Другими словами, если присвоить данному сегменту высшую нулевую привилегию, то эта привилегия будет действовать вне зависимости от того, какие уровни привилегий установлены на уровне страниц. Но если у сегмента установлен уровень привилегий равный 3, то теперь все определяется уровнем привилегий страниц, которых может быть всего два: 0 и 3.

В Windows принята следующая модель памяти — она называется плоской. В сегментные регистры загружается селектор, указывающий на дескриптор сегмента, который начинается по адресу 0. Поскольку смещение в сегменте определяется 32-битным значением, то в результате мы и получаем 4-гигабайтное адресное пространство. Других сегментов в системе не предполагается. То есть мы имеем один огромный сегмент. "Как же осуществляется защита?" — спросите вы. А защита осуществляется на уровне страниц. На рис. 3.6.4 мы видим область памяти, которая занята операционной системой (ядром, как мы его определили). Эти страницы защищены от доступа со стороны обычных исполняемых программ, хотя они находятся в том же адресном пространстве. Замечу также, что страничная схема памяти позволяет защитить код самой программы. Страницы кода программы помечаются как "только для чтения". Вместе с тем, работают все механизмы сегментной адресации — шлюзы, прерывания и т. д. — ведь все это на страничном уровне реализовать нельзя. Просто все это происходит "в тайне" от обычной программы, которая видит только плоское адресное пространство.

Как же осуществляется многозадачность? А здесь вступает в действие механизм таблиц страниц (см. рис. 3.6.3). Регистр `CR3` содержит адрес каталога таблиц страниц. Этот каталог индивидуален для каждой задачи. Переключение задач, таким образом, может осуществиться изменением содержимого регистра `CR3`. Содержимое каталога определяет отображение виртуального адресного пространства на реальные физические ресурсы компьютера. Но что очень важно: в каждое адресное пространство задачи отображается и ядро операционной системы. Наша задача — рассмотреть легальные механизмы того, как сделать, чтобы драйвер (т. е. некий программный модуль) мог быть

запущен в режиме ядра и взаимодействовал бы с приложениями, работающими с низкими привилегиями. Драйвер режима ядра обладает огромной властью над компьютером и операционной системой. Во-первых, он напрямую может обращаться к внешним устройствам через порты ввода/вывода (вот так, мы и до портов добрались!). Во-вторых, драйвер может непосредственно обращаться к коду ядра операционной системы. Эти возможности накладывают и огромную ответственность — малейшая ошибка может вызвать крах всей системы — синий экран смерти, перезапуск и т. п. Но это и волнующе. Чувствуете, как мы подходим к святая святых операционной системы Windows?

Управление драйверами

Вы будете удивлены, но драйверы, работающие в режиме ядра, управляются по той же схеме, что и сервисы. В *главе 3.8*[2] я подробно разъяснил, как устанавливать сервисы, как их запускать, останавливать и удалять. Так вот, драйверы режима ядра управляются точно так же. Программы, приведенные в *главе 3.8*, можно использовать почти без изменения. Напомню, что программа из листинга 3.8.1 — это текст самого сервиса. Разумеется, здесь он нам не понадобится. Программа из листинга 3.8.2 — установка сервиса, из листинга 3.8.3 — запуск сервиса, из листинга 3.8.4 — остановка и удаление сервиса. Из всех программ небольшому изменению следует подвергнуть только программу 3.8.2. И эти изменения коснутся только вызова функции `CreateService`. Далее представлен фрагмент, показывающий, как при помощи `CreateService` можно установить и драйвер режима ядра.

```
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH OFFSET NM
PUSH SERVICE_ERROR_NORMAL
PUSH SERVICE_DEMAND_START
PUSH SERVICE_KERNEL_DRIVER
PUSH SERVICE_START + DELETE ;вместо SERVICE_ALL_ACCESS
PUSH OFFSET SNAME1
PUSH OFFSET SNAME1
PUSH H1
CALL CreateServiceA@52
```

---

[2] Еще раз настоятельно рекомендую вернуться к *главе 3.8* и внимательно с ней поработать, иначе далее будет многое непонятно.

Я несколько видоизменил параметры. Вот значения используемых нами констант:

```
DELETE                 equ 10000h
SERVICE_START          equ 10h
SERVICE_KERNEL_DRIVER equu 00000001h
```

Особо обратите внимание на пятый параметр (снизу, разумеется). Он определяет тип сервиса. И тип этот `SERVICE_KERNEL_DRIVER`. То есть мы сообщаем менеджеру сервисов, что это не обычный сервис, а драйвер, да не простой, а работающий в режиме ядра.

> ***Замечание***
>
> Отмечу, что программа SCP, которая позволяла нам управлять сервисами в диалоговом режиме, теперь для нас бесполезна, а все воздействия на драйвер можно осуществлять только программным путем. Но что касается реестра, то данные о драйверах помещается в том же разделе, что и о службах (сервисах), т. е. в ветви HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services. Соответственно, удалив упоминания о драйвере из реестра, вы тем самым удаляете его из системы. Впрочем, посмотреть список драйверов, среди которых при правильной установке вы без труда найдете и свой драйвер, можно с помощью программы msinfo32.exe. На рис. 4.6.1 представлена консоль этой программы. К сожалению, кроме просмотра, другие полномочия у программы отсутствуют.

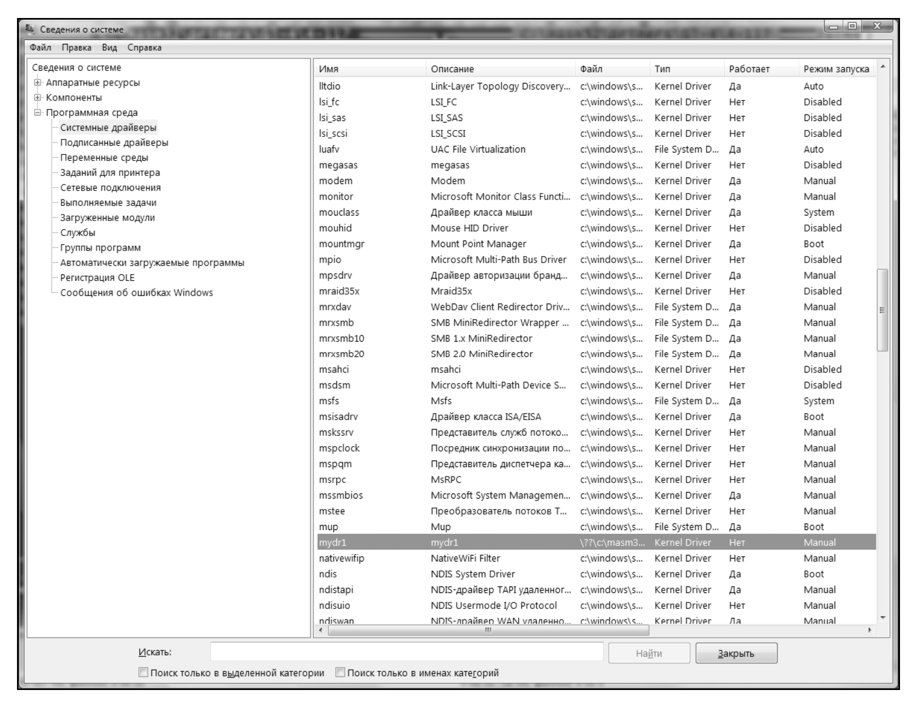


Рис. 4.6.1. Окно программы msinfo32.exe (Windows Vista) позволяет просмотреть список существующих в системе драйверов

Пример простейшего драйвера, работающего в режиме ядра

Наша задача — написать простейший драйвер режима ядра, но так, чтобы продемонстрировать основные возможности, которыми обладают программы, имеющие самые высокие привилегии. Выбор пал на воспроизведение звука. В *главе 6* книги "Assembler. Учебный курс" (см. [1]) вашего покорного слуги приводится простая программа, воспроизводящая простой звуковой сигнал с помощью встроенного динамика. Но эта программа предназначена для работы в среде операционной системы MS-DOS. Чтобы сохранить чистоту эксперимента, попробуем вначале воспроизвести алгоритм подачи сигнала, представленной в программе из указанной книги, в простой консольной программе (листинг 4.6.1).

Листинг 4.6.1. Попытка воспроизвести звук путем непосредственного обращения к портам ввода/вывода в консольной программе

```
.586P
;плоская модель
.MODEL FLAT, stdcall
EXTERN  ExitProcess@4:NEAR
;-----------------------------------------------
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;установка режима записи
      CLI
      MOV AL,10110110B
      OUT 43H,AL
      IN  AL,61H
;разрешить связь с таймером
      OR  AL,3
      OUT 61H,AL
      MOV AX,1200
;установить частоту звука
;таймер начинает действовать немедленно по засылке счетчика
      OUT 42H,AL
      MOV AL,AH
```

```
      OUT 42H,AL
      STI
      MOV ECX,0FFFFFFH
;задержка
LOO:
      LOOP LOO
;отключить канал от динамика, т. е. прекратить звук
      CLI
      IN  AL,61H
      AND AL,11111100B
      OUT 61H,AL
      STI
;-------------------------------
      PUSH 0
      CALL ExitProcess@4
_TEXT ENDS
END START
```

Алгоритм из книги [1] воспроизведен один к одному (см. листинг 4.6.1). Я увеличил только задержку с поправкой на быстродействие современных компьютеров. Данную консольную программу вы можете откомпилировать обычным для таких программ способом. Но вот беда, при ее запуске появится следующее сообщение об ошибке (рис. 4.6.2).

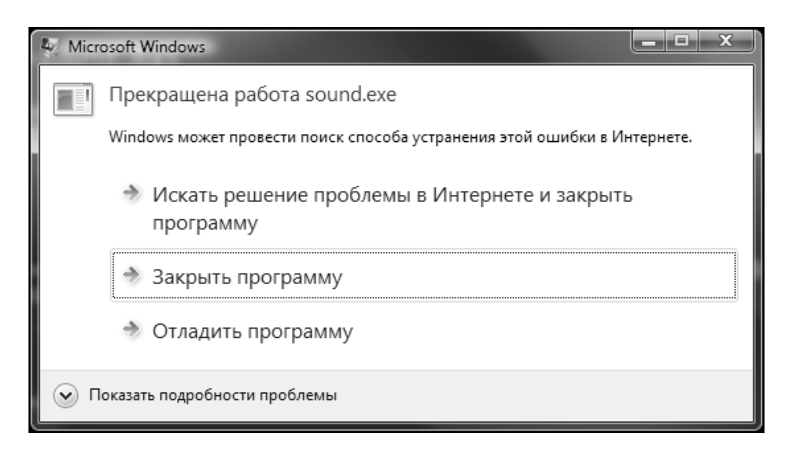


Рис. 4.6.2. Сообщение об ошибке при попытке получить доступ к защищенным портам (в операционной системе Windows Vista)

Все очень просто. Эта программа запускается в кольце 3 защиты, и операционная система сигнализирует вам о том, что вы не можете выполнять привилегированные команды ввода/вывода в этом кольце. Это первая половина

эксперимента. Во второй половине мы должны убедиться, что драйвер, работающий в режиме ядра, может воспроизвести звук указанным выше способом.

Однако давайте остановимся, и я дам пояснения по поводу работы с динамиком на низком уровне.

Пояснение по программированию звука

В основе генерации звука лежит взаимодействие микросхемы таймера и динамика. Микросхема таймера считает импульсы, получаемые от тактового генератора, и может по прошествии определенного количества импульсов (это можно запрограммировать) выдавать на выход сигнал. Если эти сигналы направить на вход динамика, то будет произведен звук. Высота его будет зависеть от частоты поступления сигналов на вход динамика. Вот кратко идея генерации звука в стандартной конфигурации IBM PC. А сейчас об этом более подробно. Микросхема таймера имеет три канала.

- Канал 0 отвечает за ход системных часов. Сигнал с этого канала вызывает прерывание времени. 18,2 раз в секунду выполняется процедура, на которую направлен вектор с номером 8. Эта процедура производит изменения в области памяти, где хранится текущее время. В специальном регистре задвижки хранится число синхроимпульсов, по прошествии которых сигнал таймера должен вызвать прерывание времени. Уменьшая это число (через порт канала), можно заставить идти системные часы быстрее. Адрес порта канала 0 — `40H`.
- Канал 1 отвечает за регенерацию памяти. Адрес порта этого канала — `41H`. В принципе, можно уменьшить число циклов регенерации памяти в секунду. Что может несколько увеличить производительность компьютера. Однако это можно сделать лишь в некоторых пределах, т. к. при увеличении промежутка регенерации возрастает вероятность сбоя памяти.
- Канал 2 обычно используется для работы с динамиком, хотя сигналы с него можно использовать и для других целей. Адрес порта этого канала — `42H`. Идея генерации звука проста. В порт `42H` посылается число (счетчик). Немедленно значение счетчика начинает уменьшаться. По достижению 0 на динамик подается сигнал. После чего процесс повторяется. Чем больше значение счетчика, тем реже подается сигнал и тем ниже звук. Связь канала 2 с динамиком устанавливается через порт `61H`. Если бит 1 этого порта установлен в 1, то канал 2 посылает сигналы на динамик. Кроме того, чтобы разрешить поступления сигнала от тактового генератора в канал 2, бит 0 этого порта должен быть равен 1 (уровень сигнала высокий). Для программирования каналов требуется вначале установить порт с адресом `43H`. Значения битов этого порта представлены в табл. 4.6.1.

Таблица 4.6.1. *Биты порта* `43H`

Бит	Значение
0	0 — двоичные данные, 1 — данные в двоично-десятичном виде
1—3	Номер режима, обычно используется режим 3

Таблица 4.6.1 *(окончание)*

Бит	Значение
4—5	Тип операции: • 00 — передать значение счетчика в задвижку; • 01 — читать/писать только старший байт; • 10 — читать/писать только младший байт; • 11 — читать/писать старший байт, потом младший
6—7	Номер программируемого канала (0—2)

Обратите внимание на использование команд `CLI` и `STI` (см. листинг 4.6.1). Они запрещают и разрешают прерывания процессора. Запрет прерываний перед работой с портами ввода/вывода — дело обычное: очень нежелательно, чтобы во время операции произошло прерывание.

Итак, с генерацией звука все понятно, и наша задача теперь — показать, что описанный выше способ генерации можно осуществить в драйвере, работающем в режиме ядра, т. е. режим ядра действительно дает нам возможность напрямую обращаться к портам ввода/вывода (листинг 4.6.2).

Листинг 4.6.2. Пример простого драйвера (sound.asm), работающего в режиме ядра и воспроизводящего короткий звуковой сигнал

```
;пример драйвера режима ядра,
;воспроизводящего короткий звуковой сигнал
.586P
.MODEL FLAT, stdcall

INCLUDE KERN.INC

includelib c:\masm32\lib\hal.lib

_TEXT SEGMENT
;DWORD PTR [EBP+0CH]  ; указывает на DRIVER_OBJECT
;DWORD PTR [EBP+08H]  ; указывает на UNICODE_STRING
ENTRY PROC ;точка входа
      PUSH EBP
      MOV  EBP,ESP   ; теперь EBP указывает на вершину стека
      PUSH EBX
      PUSH ESI
      PUSH EDI
;задание частоты
```

```
      MOV EDX,1200
;установка режима записи
      CLI
      MOV AL,10110110B
      OUT 43H,AL
      IN  AL,61H
;разрешить связь с таймером
      OR  AL,3
      OUT 61H,AL
;установить частоту звука
;таймер начинает действовать немедленно по засылке счетчика
      MOV EAX,EDX
      OUT 42H,AL
      MOV AL,AH
      OUT 42H,AL
      MOV ECX,0FFFFFFH
      STI
;задержка
LOO:
      LOOP LOO
;отключить канал от динамика, т. е. прекратить звук
      CLI
      IN  AL,61H
      AND AL,11111100B
      OUT 61H,AL
      STI
;установить код выхода
      MOV EAX,STATUS_DEVICE_CONFIGURATION_ERROR
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      RET 8
ENTRY ENDP
_TEXT ENDS
END ENTRY
```

Трансляция драйвера из листинга 4.6.2:

```
ml /c /coff sound.asm
link /driver /base:0x1000 /subsystem:native sound.obj
```

Комментарий к программе из листинга 4.6.2.

- Драйвер режима ядра не может использовать API-функции, которые применяют обычные непривилегированные программы. Ему нужны функции, доступные из ядра. Поэтому следует использовать другие библиотеки.

Эти библиотеки можно получить из пакетов DDK для Windows (Driver Development Kit, комплект для разработки драйверов), которые можно скачать с сайта фирмы Microsoft[3]. Я включил в программу одну такую библиотеку — hal.lib, хотя в нашей программе мы и не используем ни одной такой функции. К сожалению, сведения о таких функциях ядра, часто довольно отрывочные, можно получить только в пакетах DDK или у сторонних исследователей.

- Кроме библиотек в пакеты DDK входят inc-файлы, содержащие большое количество определений типов, структур, прототипов функций и констант. Помещать все это в программу, как это делали раньше, не имеет смысла. Из нескольких таких файлов я создал свой inc-файл (kern.inc), где содержатся только основные структуры и типы. В *приложении 5* приведен полный текст этого файла. Я также включаю в программу этот файл, хотя использую из него всего одну константу `STATUS_DEVICE_CONFIGURATION_ERROR`.
- Точкой входа в драйвер является процедура `ENTRY`. Процедура запускается, когда выполняется функция `StartService`: мы уже говорили о программах в листингах 3.8.2—3.8.4[4] и какие изменения надо в них внести, чтобы они обслуживали работу драйвера режима ядра. В результате будет выполнен код обращения портам ввода/вывода, и будет слышен короткий звуковой сигнал. Процедура далее возвращает значение `STATUS_DEVICE_CONFIGURATION_ERROR`. Это прием, который приводит к тому, что система после выполнения процедуры выгружается из памяти (хотя драйвер остается в базе драйверов, т. е. регистре).
- Процедура входа имеет два параметра, которые мы пока никак не использовали. Первый параметр указывает на структуру `DRIVER_OBJECT`. Эту структуру можно найти в *приложении 5*. Она громоздка, ссылается на другие структуры, а те, в свою очередь, на другие. В общем, дело довольно запутанное. Но для нас важно: данная структура описывает объект под названием "драйвер режима ядра", который мы, собственно, и создаем. Мы можем заполнять некоторые поля структуры, тем самым, устанавливая свойства объекта ядра. Второй параметр — строка в кодировке Unicode. Это строка дает название раздела реестра, где хранятся инициализации нашего драйвера.

---

[3] Впрочем, не все DDK Microsoft разрешает свободно скачивать.

[4] На прилагаемом к книге компакт-диске представлены материалы по драйверу, текст которого находится в листинге 4.6.2, располагаются в каталоге 4.6. В нем setserv.asm — программа установки драйвера, stserv.asm — программа запуска драйвера и delserv.asm — программа удаления драйвера. Текст драйвера располагается в файле mydrive1.asm.

Как видим, параметры командной строки программы ml.exe обычные. Параметры же командной строки link.exe особые. Параметр `/driver` как раз и определяет, что создается именно драйвер. Ключ `/base:0x1000` сообщает компоновщику, что загружаться драйвер будет по адресу `1000H`. Наконец `/subsystem:native` — стандартное значение подсистемы для драйверов Windows NT.

Для того чтобы стимулировать вас на дальнейшее изучение материала и одновременно закрепить пройденное, я бы предложил вам, дорогие читатели, переписать наш драйвер с использованием двух интересных функций ядра. Первая функция `READ_PORT_UCHAR` имеет всего один параметр (т. е. прототип `READ_PORT_UCHAR@4`). Этим параметром является адрес порта ввода/вывода. Эта функция фактически заменяет команду микропроцессора `IN`. Вторая функция `WRITE_PORT_UCHAR` имеет два параметра. Первый параметр — номер порта ввода/вывода, второй параметр — операнд, из которого данные будут переданы в порт. Эти функции находятся в библиотеке hal.lib, а их прототипы, как обычно, вы должны определить в начале вашей программы. Вот, собственно и все. Результат должен быть таким же, как при использовании стандартных команд `IN` и `OUT` микропроцессора Intel. Может, правда, возникнуть вопрос: для чего, собственно, такое дублирование. Очевидно, для совместимости, когда операционная система будет работать с микропроцессорами другого семейства.

> ***Замечание***
>
> Вообще звуковой сигнал — это довольно удобный способ отладки драйверов. Другие способы сигнализации о том, как выполнилась та или иная функция, не столь просты в употреблении или даже опасны.

Драйверы режима ядра и устройства

Задача этого раздела — написать еще один драйвер режима ядра, создающего устройство, к которому можно обращаться через стандартный файловый ввод/вывод.

Обратимся вначале к программе, работающей в пользовательском режиме и обращающейся к устройству, созданному драйвером режима ядра, т. е. фактически к самому драйверу (листинг 4.6.3). Данная программа является четвертой программой управления драйверами. Три первых программы установки, запуска и удаления драйвера обсуждались нами неоднократно и фактически совпадают с программами управления сервисами. Устройство, созданное и обслуживаемое драйвером, открывается с помощью универсальной функции `CreateFile`. Обратите внимание на имя устройства: `'\\.\SLN'`.

При успешном открытии программа посылает драйверу (через устройство) данные с помощью функции `DeviceIoControl` и при успешном ее выполнении получает данные из драйвера. Для проверки используется функция `MessageBox`, с помощью которой эти данные выводятся. Для закрытия устройства вызывается вполне стандартная функция `CloseHandle`, применяемая для закрытия большинства создаваемых объектов ядра.

Листинг 4.6.3. Программа, открывающая устройство, созданное драйвером (см. листинг 4.6.4)

```
.586P
;плоская модель
.MODEL FLAT, stdcall

DTP    equ 8000H
ACCESS equ 0H
OPER   equ 800H
MBUF   equ 0H
;передаваемая в драйвер команда
CMD    equ (DTP SHL 16) OR (ACCESS SHL 14) OR (OPER SHL 2) OR MBUF

GENERIC_READ      equ 80000000h
GENERIC_WRITE     equ 40000000h
GEN = GENERIC_READ or GENERIC_WRITE
SHARE = 0
OPEN_EXISTING     equ 3
STD_OUTPUT_HANDLE equ -11

EXTERN  CloseHandle@4:NEAR
EXTERN  CreateFileA@28:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  MessageBoxA@16:NEAR
EXTERN  MessageBoxW@16:NEAR
EXTERN  wsprintfA:NEAR
EXTERN  GetLastError@0:NEAR
EXTERN  lstrlenA@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  GetStdHandle@4:NEAR
EXTERN  DeviceIoControl@32:NEAR
;------------------------------------------------
;директивы компоновщику для подключения библиотек
includelib d:\masm32\lib\kernel32.lib
includelib d:\masm32\lib\user32.lib
;------------------------------------------------

;сегмент данных
```

```
_DATA SEGMENT
;имя устройства
      PATH    DB '\\.\SLN',0
      H1      DD ?
      HANDL   DD ?
      LENS    DD ?
      BUF1    DB 512 DUP(0)
      ERRS    DB "Error %u ",0
      BUFIN   DB "For Mydriver",88 DUP(0)
      BUFOUT  DB 100 DUP(0)
      BYTES   DD 0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
      PUSH STD_OUTPUT_HANDLE
      CALL GetStdHandle@4
      MOV  HANDL,EAX
;открыть устройство
      PUSH 0
      PUSH 0
      PUSH OPEN_EXISTING
      PUSH 0
      PUSH 0
      PUSH GEN
      PUSH OFFSET PATH
      CALL CreateFileA@28
      CMP  EAX,-1
      JNZ  NOER
      CALL ERROB
      JMP  EXI
NOER:
;передать в драйвер команду и данные в буфере BUFIN
      MOV  H1,EAX
      PUSH 0
      PUSH OFFSET BYTES
      PUSH 100
      PUSH OFFSET BUFOUT
      PUSH 100
      PUSH OFFSET BUFIN
      PUSH CMD
      PUSH H1
      CALL DeviceIoControl@32
      CMP  EAX,0
      JZ   CLOS
      CMP  BYTES,0
```

```
      JZ   CLOS
;вывести содержимое возвращенного буфера
      PUSH 0
      PUSH OFFSET BUFOUT
      PUSH OFFSET BUFOUT
      PUSH 0    ; дескриптор экрана
      CALL MessageBoxA@16
CLOS:
      PUSH  H1
      CALL  CloseHandle@4
EXI:
      PUSH 0
      CALL ExitProcess@4
ERROB:
      CALL GetLastError@0
      PUSH EAX
      PUSH OFFSET ERRS
      PUSH OFFSET BUF1
      CALL wsprintfA
      ADD  ESP,12
      LEA  EAX,BUF1
      MOV  EDI,1
      CALL WRITE
      RET
;вывести строку (в конце - перевод строки)
;EAX - на начало строки
;EDI - с переводом строки или без
WRITE   PROC
;получить длину параметра
        PUSH  EAX
        PUSH  EAX
        CALL  lstrlenA@4
        MOV   ESI,EAX
        POP   EBX
        CMP   EDI,1
        JNE   NO_ENT
;в конце - перевод строки
        MOV  BYTE PTR [EBX+ESI],13
        MOV  BYTE PTR [EBX+ESI+1],10
        MOV  BYTE PTR [EBX+ESI+2],0
        ADD  EAX,2
NO_ENT:
;вывод строки
        PUSH 0
        PUSH OFFSET LENS
        PUSH EAX
```

```
        PUSH EBX
        PUSH HANDL
        CALL WriteConsoleA@20
        RET
WRITE   ENDP
_TEXT ENDS
END START
```

Трансляция программы из листинга 4.6.3:

```
ml /c /coff prog.asm
link /subsystem:console prog.obj
```

Прокомментирую программу из листинга 4.6.3.

- Функцию `DeviceIoControl` мы уже разбирали в данной главе. Но я хотел бы обратить внимание на второй параметр функции. Здесь помещается передаваемая в драйвер инструкция (control code). Существует набор управляющих кодов для драйверов определенных устройств. При создании своего управляющего кода следует иметь в виду, что он образуется по определенным правилам. Вот эти простые правила. Биты 0—1 определяют, будет ввод/вывод буферизованным или нет. Значение 0 — стандартный буферизованный ввод/вывод, другие значения определяют небуферизованный ввод/вывод. Биты 3—13 определяют команду, которую драйвер должен выполнить. Значения `0—7FFH` зарезервированы, остальные могут использоваться программистами по своему усмотрению. Биты 14—15 определяют запрашиваемые права доступа. Значение 0 — максимально возможные права доступа. Значения 1 и 2 доступ, соответственно, на чтение и запись. Последние биты 16—31 — определяют тип устройства. Здесь программисты по своему усмотрению могут использовать значения в диапазоне `8000H—0FFFFH`. Мы образовали свою собственную команду `CMD` (см. листинг 4.6.3).
- Отладка драйвера режима ядра — довольно кропотливая работа. Синего экрана вам не избежать. В процессе отладки участвуют и программы, взаимодействующие с драйвером. Очень важную роль здесь играет обработка ошибок. Мы обрабатываем и выводим возможные ошибки функции `CreateFile`. Причину ошибок легко получить с помощью программы errlookup.exe, о которой я уже неоднократно упоминал в книге. Функция `MessageBox` также играет роль отладочного механизма не только данной программы, но и драйвера, с которым осуществляется взаимодействие.

Обратимся теперь к тексту драйвера, представленному в листинге 4.6.4. Как и простейший драйвер, описанный выше, данный драйвер содержит процедуру входа (entry). Эта процедура выполняет следующие задачи:

- создать устройство;

- создать символьную ссылку на устройство. Именно это имя используется в качестве имени устройства в функции `CreateFile`;
- определить процедуру выгрузки драйвера. Эта процедура вызывается перед удалением драйвера. Здесь следует удалить или освободить все выделенные драйвером ресурсы;
- определить процедуры откликов. Мы определяем три такие процедуры: вызываемая при открытии устройства (`CreateFile`), вызываемая при закрытии устройства (`CloseHandle`), вызываемая при отправке устройству управляющего кода.

Как уже было сказано, функции, работающие в ядре, оперируют строками в кодировке Unicode. В частности, нам придется иметь дело со следующими структурами:

```
UNICODE_STRING STRUCT
      woLength        WORD   ?
      MaximumLength   WORD   ?
      Buffer          PWSTR  ?
UNICODE_STRING ENDS
```

Здесь `woLength` — длина Unicode-строки в байтах, `MaximumLength` — длина буфера в байтах, где содержится строка, `Buffer` — адрес буфера (`PWSTR` — это просто `PTR WORD`). Для заполнения этой структуры на основе заранее подготовленной в кодировке Unicode строки используется функция `_RtlInitUnicodeString`[5].

В листинге 4.6.4 представлен текст драйвера, который при инициализации создает устройство, к которому затем можно обратиться из прикладной программы.

Листинг 4.6.4. Драйвер режима ядра, создающий устройство с обработкой нескольких запросов

```
.586P
.MODEL FLAT

INCLUDE KERN.INC

EXTERN _RtlInitUnicodeString@8:NEAR
EXTERN _IoCreateDevice@28:NEAR
EXTERN _IoCreateSymbolicLink@8:NEAR
EXTERN _IoDeleteSymbolicLink@4:NEAR
```

[5] Надеюсь, вы не забыли, что на уровне ядра мы не можем использовать обычные API-функции, а используем специальные, которые действуют только в пределах ядра.

```
EXTERN _IoDeleteDevice@4:NEAR
EXTERN @IofCompleteRequest@8:NEAR

DTP    equ 8000H
ACCESS equ 0H
OPER   equ 800H
MBUF   equ 0H
;команда, обрабатываемая драйвером
CMD    equ (DTP SHL 16) OR (ACCESS SHL 14) OR (OPER SHL 2) OR MBUF

includelib c:\masm32\lib\ntoskrnl.lib

_DATA SEGMENT
;имя устрйоства
      DNAME DW "\","D","e","v","i","c","e","\","D","e","v",0
;ссылка на имя
      LNAME DW "\","?","?","\","S","L","N",0
;структуры UNICODE_STRING для хранения имен
      DEVN UNICODE_STRING   <?>
      SLNKN UNICODE_STRING  <?>
;структура, используемая для создания устройства
      PD DEVICE_OBJECT <?>
;другие переменные
      MES  DB "Из глубины",0
      MES1 DB "For Mydriver",0
_DATA ENDS

_TEXT SEGMENT

;процедура входа в драйвер
;DWORD PTR [EBP+08H] — указатель на структуру DRIVER_OBJECT
;DWORD PTR [EBP+12] — указатель на структуру UNICODE_STRING
_ENTRY PROC ;точка входа драйвера
PUSH EBP
      MOV EBP,ESP ;теперь EBP указывает на вершину стека
      PUSH EBX
      PUSH ESI
      PUSH EDI
;определить имя устройства
      PUSH OFFSET LNAME
      PUSH OFFSET SLNKN
      CALL _RtlInitUnicodeString@8
;определить символьную ссылку на имя устройства
      PUSH OFFSET DNAME
      PUSH OFFSET DEVN
      CALL _RtlInitUnicodeString@8
```

```
;создать устройство
      PUSH OFFSET PD
      PUSH 0
      PUSH 0
      PUSH 22H ;FILE_DEVICE_UNKNOWN
      PUSH OFFSET DEVN
      PUSH 0
      PUSH DWORD PTR [EBP+08H]
      CALL _IoCreateDevice@28
      CMP  EAX,0
      JZ   OK
      MOV  EAX, STATUS_DEVICE_CONFIGURATION_ERROR
      JMP EXI
OK:
;создать символьную ссылку на устройство
      PUSH OFFSET DEVN
      PUSH OFFSET SLNKN
      CALL _IoCreateSymbolicLink@8
;EAX на структуру DRIVER_OBJECT
      MOV EAX,DWORD PTR [EBP+08H]
      ASSUME EAX:PTR DRIVER_OBJECT
;определяем процедуру выгрузки драйвера
      MOV [EAX].DriverUnload, OFFSET DELDRIVER
;определяем процедуры откликов
;открытие устройства
      MOV [EAX].MajorFunction[IRP_MJ_CREATE*4], OFFSET CR_FILE
;закрытие устройства
      MOV [EAX].MajorFunction[IRP_MJ_CLOSE*4],  OFFSET CL_FILE
;управление устройством
      MOV [EAX].MajorFunction[IRP_MJ_DEVICE_CONTROL*4], OFFSET CTL_DEV
      ASSUME EAX:NOTHING
;установить код выхода
      MOV EAX,STATUS_SUCCESS
EXI:
      POP EDI
      POP ESI
      POP EBX
      POP EBP
      RET 8
_ENTRY ENDP
;процедура, вызываемая при удалении драйвера
;DWORD PTR [EBP+08H] - указатель на структуру DRIVER_OBJECT
DELDRIVER PROC
PUSH EBP
      MOV  EBP,ESP ;теперь EBP указывает на вершину стека
;удалить символьную ссылку
```

```
      PUSH OFFSET SLNKN
      CALL _IoDeleteSymbolicLink@4
;удалить устройство
      MOV  EAX,DWORD PTR [EBP+08H]
      ASSUME EAX:PTR DRIVER_OBJECT
      PUSH [EAX].DeviceObject
      CALL _IoDeleteDevice@4
      POP  EBP
      RET  4
DELDRIVER ENDP
;процедура, вызываемая при выполнении CreateFile
;DWORD PTR [EBP+08H] - указатель на структуру DEVICE_OBJECT
;DWORD PTR [EBP+12] - указатель на структуру IRP
CR_FILE PROC
      PUSH EBP
      MOV  EBP,ESP
      MOV  EAX,DWORD PTR [EBP+12]
      ASSUME EAX:PTR _IRP
      MOV  [EAX].IoStatus.Status,STATUS_SUCCESS
      MOV  [EAX].IoStatus.Information,0
      ASSUME EAX:NOTHING
;завершение операции ввода/вывода
;вызов в формате fastcall
      MOV  ECX,DWORD PTR [EBP+12]
      XOR  EDX,EDX
      CALL @IofCompleteRequest@8
      MOV  EAX,STATUS_SUCCESS
      POP  EBP
      RET  8
CR_FILE ENDP
;процедура, вызываемая при выполнении CloseHandle
;DWORD PTR [EBP+08H] - указатель на структуру DEVICE_OBJECT
;DWORD PTR [EBP+12] - указатель на структуру IRP
CL_FILE PROC
      PUSH EBP
      MOV  EBP,ESP
      MOV  EAX,DWORD PTR [EBP+12]
      ASSUME EAX:PTR _IRP
      MOV  [EAX].IoStatus.Status,STATUS_SUCCESS
      AND  [EAX].IoStatus.Information,0
      ASSUME EAX:NOTHING
;завершение операции ввода/вывода
;вызов в формате fastcall
      MOV  ECX,DWORD PTR [EBP+12]
      XOR  EDX,EDX
      CALL @IofCompleteRequest@8
```

```
      MOV  EAX,STATUS_SUCCESS
      POP  EBP
      RET  8
CL_FILE ENDP
;процедура, вызываемая при выполнении ControlService
;DWORD PTR [EBP+08H] - указатель на структуру DEVICE_OBJECT
;DWORD PTR [EBP+12] - указатель на структуру IRP
CTL_DEV PROC
      PUSH  EBP
      MOV   EBP,ESP
      PUSH  EBX
;EAX будет указывать на структуру IRP
      MOV   EAX,DWORD PTR [EBP+12]
      ASSUME EAX:PTR _IRP
;EDI будет указывать на структуру IO_STACK_LOCATION,
;где, в частности, должна содержаться присланная команда
      MOV EDI, [EAX].Tail.Overlay.CurrentStackLocation
      ASSUME EDI:PTR IO_STACK_LOCATION
      CMP [EDI].Parameters.DeviceIoControl.IoControlCode,CMD
      JNZ  NO_CMD
;команда на лицо
      MOV  ESI,[EAX].AssociatedIrp.SystemBuffer
;теперь проверим пришедшее сообщение
      PUSH ESI
      PUSH OFFSET MES1
      CALL CMPSTR
      CMP  EBX,1
      JZ   NO_CMD
;скопируем в буфер передаваемую из драйвера строку
      PUSH OFFSET MES
      PUSH ESI
      CALL COPYSTR
;длина строки будет также передана в прикладную программу
      PUSH OFFSET MES
      CALL LENSTR
      MOV  [EAX].IoStatus.Status,STATUS_SUCCESS
      MOV  [EAX].IoStatus.Information,EBX
NO_CMD:
      ASSUME EDI:NOTHING
      ASSUME EAX:NOTHING
;завершение операции ввода/вывода
;вызов в формате fastcall
      MOV  ECX,DWORD PTR [EBP+12]
      XOR  EDX,EDX
      CALL @IofCompleteRequest@8
      MOV  EAX,STATUS_SUCCESS
```

```
      POP  EBX
      POP  EBP
      RET  8
CTL_DEV ENDP
;процедура копирования одной строки в другую
;строка, куда копировать [EBP+08H] - первый параметр
;строка, что копировать [EBP+0CH] - второй параметр
;не учитывает длину строки, куда производится копирование
COPYSTR PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH ESI
      PUSH EDI
      PUSH EAX
      MOV  ESI,DWORD PTR [EBP+0CH]
      MOV  EDI,DWORD PTR [EBP+08H]
L1:
      MOV  AL,BYTE PTR [ESI]
      MOV  BYTE PTR [EDI],AL
      CMP  AL,0
      JE   L2
      INC  ESI
      INC  EDI
      JMP  L1
L2:
      POP  EAX
      POP  EDI
      POP  ESI
      POP  EBP
      RET  8
COPYSTR ENDP
;функция получения длины
;[EBP+08H] - указатель на строку
;в EBX - длина строки,
LENSTR PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH ESI
      MOV  ESI,DWORD PTR [EBP+8]
      XOR  EBX,EBX
LBL1:
      CMP  BYTE PTR [ESI],0
      JZ   LBL2
      INC  EBX
      INC  ESI
      JMP  LBL1
```

```
LBL2:
      POP  ESI
      POP  EBP
      RET  4
LENSTR ENDP
;процедура сравнения двух строк
;[EBP+08H] - первый параметр
;[EBP+012] - второй параметр
;EBX = 0 - строки равны, EBX = 1 - строки не равны
CMPSTR PROC
      PUSH EBP
      MOV  EBP,ESP
      PUSH ESI
      PUSH EDI
      PUSH EAX
      MOV  ESI,DWORD PTR [EBP+012]
      MOV  EDI,DWORD PTR [EBP+08H]
L1:
      MOV  AL,BYTE PTR [ESI]
      MOV  EBX,1
      CMP  BYTE PTR [EDI] ,AL
      JNZ  EXI
      XOR  EBX,EBX
      CMP  AL,0
      JZ   EXI
      INC  ESI
      INC  EDI
      JMP  L1
EXI:
      POP  EAX
      POP  EDI
      POP  ESI
      POP  EBP
      RET  8
CMPSTR ENDP
_TEXT ENDS
END _ENTRY
```

Трансляция текста драйвера из листинга 4.6.4:

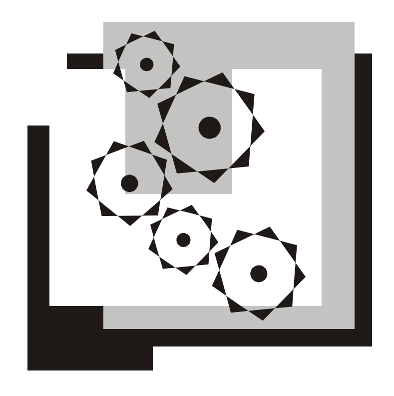
```
ML /c /coff driver.asm
link /driver /base:0x1000 /subsystem:native driver.obj
```

Комментарий к листингу 4.6.4.

- Обратите внимание на то, что в листинге отсутствует директива `stdcall`. Это не ошибка. Дело в том, что имя функции `@IofCompleteRequest` не должно

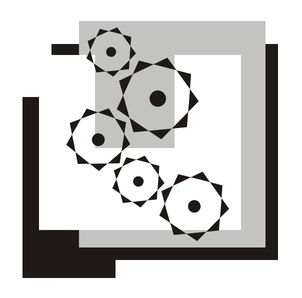
иметь префикс "_" (подчеркивание) в объектном модуле. Остальные же функции должны иметь такой префикс. По этой причине мы явно указываем этот префикс в именах функций и опустили директиву `stdcall` в начале программы. Остановимся еще на функции `@IofCompleteRequest`, которая оповещает, что запрос обработан. Функция вызывается в формате fastcall. Два параметра этой функции помещаются в регистры `ECX` и `EDX`.

- В листинге используются указатели на некоторые структуры, описание которых можно найти в *приложении 5*. Дадим их краткую характеристику.
  - `device_object` — структура, в которой описано созданное драйвером устройство (объект). Обратите внимание, что в процедуре `DELDRIVER` мы получаем указатель на эту структуру из структуры `driver_object` (`DeviceObject`).
  - `driver_object` — структура, описывающая объект-драйвер. В частности, в состав структуры входит массив `MajorFunction`. Этот массив содержит указатели на процедуры обработки запросов. Каждый элемент массива отвечает за свой запрос, а номера элементов определяются числовыми константами, указанными в заголовочных файлах (см. листинг 4.6.4).
  - Структура `_IRP` содержит информацию о запросе к драйверу (IRP — I/O request packet, т. е. пакет запроса ввода/вывода). В частности, в этой структуре будет содержаться указатель на структуру `IO_STACK_LOCATION`, где будет находиться код команды, которая пришла вместе с запросом драйверу (см. текст процедуры `CTL_DEV`).
- Обратите внимание на три процедуры: `CMPSTR` (сравнение двух строк), `LENSTR` (длина строки), `COPYSTR` (копирование одной строки в другую), появление которых обусловлено не только моим желанием поупражняться в языке ассемблера, но и не возможность использовать стандартные API-функции.



ПРИЛОЖЕНИЯ

Приложение 1



Справочник API-функций и сообщений Windows

В силу ограниченного объема книги невозможно дать полный список API-функций — их насчитывается более двух тысяч. В данном приложении представлен список API-функций, которые содержатся в данной книге, с кратким комментарием и указанием глав, где они были использованы или хотя бы упомянуты. Вторая таблица посвящена сообщениям Windows.

В третьей колонке табл. П1.1 указываются не вообще все места, где упоминается данная функция, а места, где разъясняется смысл этой функции либо просто упоминается (если нет другой информации), либо упоминается в первый раз.

Таблица П1.1. *Функции API*

Название функции	**Назначение функции**	**Где существенным образом упоминается**
`accept`	Прием запросов от программ-клиентов (connect)	*Глава 3.4*, см. листинг 3.4.5
`AllocConsole`	Создать консоль	*Глава 2.3*
`Arc`	Нарисовать дугу	Упоминание в *главе 2.1*
`BeginPaint`	Получить контекст при получении сообщения `WM_PAINT`	*Глава 2.1*, см. листинг 2.1.1
`bind`	Подключить сокет к коммуникационной среде	*Глава 3.4*, см. листинги 3.4.5 и 3.4.6
`BitBlt`	Скопировать виртуальную прямоугольную область в окно	*Глава 2.1*, см. листинг 2.1.6

Таблица П1.1 *(продолжение)*

Название функции	**Назначение функции**	**Где существенным образом упоминается**
`CallNextHookEx`	Продолжить выполнение других фильтров	*Глава 3.6, см. разд. "Фильтры (HOOKS)"*
`CallWindowProc`	Вызвать процедуру окна	*Глава 3.5*, листинг 3.5.3 и информация, которая ему предшествует
`CancelIO`	Отменить все запросы на асинхронный ввод/вывод для данного устройства, выданные данным потоком	*Глава 2.7*
`CharToOem`	Функция перекодирует строку из кодировки для графических окон в кодировку для консольных окон	*Глава 1.6*. *Глава 2.2*, см. листинг 2.2.3
`CloseHandle`	Закрыть объект: файл, консоль, коммуникационный канал, созданный функциями `CreateFile`, `CreatePipe` и т. п.	*Глава 2.6*
`CloseServiceHandle`	Закрыть дескриптор базы данных сервисов	*Глава 3.8*, см. листинги 3.8.2—3.8.4
`closesocket`	Закрыть сокет	*Глава 3.4*, см. листинги 3.4.5 и 3.4.6
`connect`	Попытка соединится с программой-сервером (см. `accept`)	*Глава 3.4*, см. листинг 3.4.5
`ConnectNamedPipe`	Переводит процесс, создавший именованный канал, в состояние ожидания подключения к нему клиентских процессов	*Глава 2.8*
`ControlService`	Отправить команду сервису	*Глава 3.8*, см. листинг 3.8.4
`CreateCompatibleBitmap`	Создать карту битов, совместимую с заданным контекстом	*Глава 2.1*, см. листинг 2.1.6

Таблица П1.1 *(продолжение)*

Название функции	Назначение функции	Где существенным образом упоминается
`CreateCompatibleDC`	Создать контекст, совместимый с данным окном	*Глава 2.1*, см. листинг 2.1.6
`CreateDialogParam`	Создать немодальное диалоговое окно	*Глава 2.4*, см. листинг 2.4.3
`CreateDirectory`	Создать каталог	*Глава 2.6*. Более подробно *см. главу 2.8*
`CreateEvent`	Создать событие	*Глава 3.2*, *см. разд. "События"*
`CreateFile`	Создать или открыть файл, консоль, коммуникационный канал или другое устройство	*Глава 2.6*. Подробное описание функции *см. в главе 2.8*
`CreateFileMapping`	Создать отображаемый файл	*Глава 3.5*, см. листинг 3.5.2
`CreateFont`	Задать параметры шрифта	Упоминается в *главе 2.1*
`CreateFontIndirect`	Задать параметры шрифта	*Глава 2.1*
`CreateMutex`	Создать объект синхронизации "взаимоисключение"	*Глава 3.2*, *см. разд. "Взаимоисключения"*
`CreateNamePipe`	Создать именованный канал	*Глава 2.8*
`CreatePen`	Создать перо	*Глава 2.1*, см. листинг 2.1.6
`CreatePipe`	Создать канал обмена информацией	*Глава 3.5*, см. листинг 3.5.4.
`CreateProcess`	Создать новый процесс	*Глава 3.2*, см. листинг 3.2.1
`CreateSemaphor`	Создать семафор	*Глава 3.2*, *см. разд. "Семафоры"*
`CreateService`	Служит для помещения сервиса в сервисную базу	*Глава 3.8*, см. листинг 3.8.2
`CreateSolidBrush`	Определить кисть	*Глава 2.1*, см. листинг 2.1.1

Таблица П1.1 *(продолжение)*

Название функции	**Назначение функции**	**Где существенным образом упоминается**
`CreateThread`	Создать поток	*Глава 3.2*, см. листинги 3.2.1 и 3.2.3
`CreateWindow`	Создать окно	*Глава 1.2*
`CreateWindowEx`	Расширенное создание окна	*Глава 1.2*
`DefWindowProc`	Вызывается для сообщений, которые не обрабатываются функцией окна	*Глава 1.2*, см. листинг 1.2.1
`DeleteCriticalSection`	Удалить объект "критическая секция"	*Глава 3.2*, *см. разд. "Критические секции"*
`DeleteDC`	Удалить контекст, полученный посредством функций типа `CreatePen` или `CreateDC`	*Глава 2.1*, см. листинг 2.1.6
`DeleteFile`	Удалить файл	*Глава 2.8*
`DeleteObject`	Удалить объект, выбранный функцией `SelectObject`	*Глава 2.1*, см. листинг 2.1.4
`DeleteService`	Удалить сервис	*Глава 3.8*, см. листинг 3.8.4
`DestroyMenu`	Удалить меню из памяти	*Глава 2.5*, см. листинг 2.5.1
`DestroyWindow`	Удалить окно из памяти	*Глава 2.4*, см. листинг 2.4.3
`DeviceIoControl`	Функция управления устройствами. Посылает управляющий код (команду) непосредственно драйверу устройства	*Главы 2.6*, *2.8* и *4.6*. См. листинг 4.6.3
`DialogBox`	Создать модальное диалоговое окно	*Глава 2.4*
`DialogBoxParam`	Создать немодальное диалоговое окно	*Глава 2.4*, см. листинг 2.4.1
`DispatchMessage`	Вернуть управление Windows с передачей сообщения, предназначенного окну	*Глава 1.2*

Таблица П1.1 *(продолжение)*

Название функции	Назначение функции	Где существенным образом упоминается
`Ellipse`	Рисовать эллипс	*Глава 2.1*
`EndDialog`	Удалить модальное диалоговое окно	*Глава 2.4*, см. листинг 2.4.1
`EndPaint`	Удалить контекст, полученный при помощи `BeginPaint`	*Глава 2.1*, см. листинг 2.1.1
`EnterCriticalSection`	Войти в критическую секцию	*Глава 3.2*, *см. разд. "Критические секции"*
`EnumProcesses`	Функция используется для получения идентификаторов процессов в системе	*Глава 3.5*, см. листинг 3.79
`EnumWindows`	Пересчитать окна	*Глава 3.5*, см. листинг 3.5.7
`ExitProcess`	Закончить данный процесс со всеми подзадачами (потоками)	*Глава 1.2*
`ExitThread`	Выход из потока с указанием кода выхода	*Глава 3.2*
`FileTimeToLocalFileTime`	Преобразовать файловое время к местному	*Глава 2.6*, см. листинг 2.5.6
`FileTimeToSystemTime`	Преобразовать структуру файлового времени в структуру `SYSTIME`, более удобную для использования	*Глава 2.6*, см. листинг 2.6.6
`FindFirstFile`	Первый поиск файлов в каталоге	*Глава 2.6*, *см. разд. "Поиск файлов"*
`FindNextFile`	Осуществить последующий поиск в каталоге	*Глава 2.6*, *см. разд. "Поиск файлов"*
`FlushViewOfFile`	Сохранить отображаемый файл или его часть на диск	*Глава 3.5*
`FreeConsole`	Освободить консоль	*Глава 2.3*, см. листинг 2.3.2
`FreeLibrary`	Выгрузить динамическую библиотеку	*Глава 3.3*, см. листинг 3.3.2
`GdipCreatePen`	Создать перо в библиотеке GDI+	*Глава 2.2*, см. листинг 2.2.1

Таблица П1.1 *(продолжение)*

Название функции	**Назначение функции**	**Где существенным образом упоминается**
`GdipCreateSolidFill`	Создать кисть в библиотеке GDI+	*Глава 2.2*, см. листинг 2.2.1
`GdipDeleteBrush`	Удалить кисть в библиотеке GDI+	*Глава 2.2*, см. листинг 2.2.1
`GdipDeletePen`	Удалить перо в библиотеке GDI+	*Глава 2.2*, см. листинг 2.2.1
`GdipDrawLineI`	Рисовать в библиотеке GDI+	*Глава 2.2*, см. листинг 2.2.1
`GdipFillEllipseI`	Нарисовать закрашенный эллипс в библиотеке GDI+	*Глава 2.2*, см. листинг 2.2.1
`GdiplusStartup`	Инициализация библиотеки GDI+	*Глава 2.2*, см. листинг 2.2.1
`GetCommandLine`	Получить командную строку программы	*Глава 2.3*
`GetCurrentDirectory`	Получить имя текущего каталога	Подробно *см. главу 2.8*
`GetCursorPos`	Получить положение курсора в экранных координатах	*Глава 3.1*, см. листинг 3.1.3
`GetDC`	Получить контекст окна	*Глава 2.1*, см. листинг 2.1.6
`GetDiskFreeSpace`	Определить объем свободного пространства на диске	*Глава 3.4*, см. комментарий после рис. 3.4.1
`GetDlgItem`	Получить дескриптор управляющего элемента в окне	*Глава 3.1*, см. листинг 3.1.3
`GetDriveType`	Получить тип устройства	*Глава 3.4*, см. листинг 3.4.1
`GetFileTime`	Получить временные характеристики файла	*Глава 2.6*, см. листинг 2.6.6
`GetFullPathName`	Преобразовать короткое имя файла в длинное имя	*Глава 2.6*, *см. разд. "Характеристики файлов"*
`gethostbyname`	Получить данные об узле по его имени	*Глава 3.4*, см. листинг 3.4.6

Таблица П1.1 *(продолжение)*

Название функции	**Назначение функции**	**Где существенным образом упоминается**
`GetLastError`	Получить последнюю ошибку — ошибку, происшедшую при выполнении последней API-функции	*Глава 1.2*, см. также листинг 2.8.3
`GetLocalTime`	Получить местное время	*Глава 3.1*, см. листинг 3.1.2
`GetMenuItemInfo`	Получить информацию о выбранном пункте меню	*Глава 2.5*, см. листинг 2.5.1
`GetMessage`	Получить следующее сообщение из очереди сообщений данного приложения	*Глава 1.2*
`GetModuleFileNameEx`	Функция используется для получения полного имени какого-либо модуля процесса	*Глава 3.5*, см. листинг 3.5.8
`GetModuleHandle`	Получить дескриптор приложения	*Глава 1.2*, см. листинг 1.2.2
`GetProcAddress`	Получить адрес процедуры (в динамической библиотеке)	*Глава 3.3*
`GetShortPathName`	Преобразовать длинное имя файла в короткое имя	*Глава 2.6*, *см. разд. "Характеристики файлов"*
`GetStdHandle`	Получить дескриптор консоли	*Глава 2.3*
`GetStockObject`	Определить дескриптор стандартного объекта	*Глава 2.1*
`GetSystemDirectory`	Получить системный каталог	*Глава 3.2*
`GetSystemMetrics`	Определить значение системных характеристик	*Глава 2.1*, см. листинг 2.1.6
`GetSystemTime`	Получить время по Гринвичу	Упоминается в *главе 3.1*
`GetTextExtentPoint32`	Определить параметры текста в данном окне	*Глава 2.1*, см. листинг 2.1.2
`GetVolumeInformation`	Получить информацию о типе файловой системы данного раздела	*Глава 2.6*

Таблица П1.1 *(продолжение)*

Название функции	Назначение функции	Где существенным образом упоминается
`GetWindowRect`	Определить размер окна	*Глава 2.1*, см. листинг 2.1.2
`GetWindowsDirectory`	Получить каталог Windows	*Глава 3.2*
`GetWindowsRec`	Получить координаты окна	*Глава 3.1*
`GetWindowText`	Получить заголовок окна	*Глава 3.5*, см. листинг 3.5.7
`GlobalAlloc`	Выделить блок памяти	*Глава 3.6*, см. листинг 3.6.1
`GlobalDiscard`	Удалить удаляемый блок памяти	*Глава 3.6*
`GlobalFree`	Освободить блок памяти	*Глава 3.6*, см. листинг 3.6.1
`GlobalLock`	Фиксировать перемещаемый блок памяти	*Глава 3.6*, *см. разд. "Динамическая память"*
`GlobalReAlloc`	Изменить размер блока памяти	*Глава 3.6*, *см. разд. "Динамическая память"*
`GlobalUnLock`	Снять фиксацию блока памяти	*Глава 3.6*, *см. разд. "Динамическая память"*
`InitializeCriticalSection`	Создать объект "критическая секция"	*Глава 3.2*, см. листинг 3.2.3
`InvalidateRect`	Перерисовать окно	*Глава 2.1*, см. листинг 2.1.6
`KillTimer`	Удалить таймер	Глава 3.1. см. листинг 3.1.1
`LeaveCriticalSection`	Покинуть критическую секцию	*Глава 3.2*, см. листинг 3.2.3
`LineTo`	Провести линию от текущей точки к заданной	*Глава 2.1*, см. листинг 2.1.6

Таблица П1.1 *(продолжение)*

Название функции	Назначение функции	Где существенным образом упоминается
`Listen`	Перевести сокет в состояние, в котором он слушает внешние вызовы	*Глава 3.4*, см. листинг 3.4.5
`LoadAccelerators`	Загрузить таблицу акселераторов	*Глава 2.4*, см. листинг 2.4.3
`LoadCursor`	Загрузить системный курсор или курсор, определенный в файле ресурсов	*Глава 1.2*
`LoadIcon`	Загрузить системную пиктограмму или пиктограмму, определенную в файле ресурсов	*Глава 1.2*
`LoadLibrary`	Загрузить динамическую библиотеку	*Глава 3.3*, см. листинг 3.3.2
`LoadMenu`	Загрузить меню, которое определено в файле ресурсов	*Глава 2.4*, см. листинг 2.4.3
`LoadString`	Загрузить строку, определенную в файле ресурсов	*Глава 2.4*, см. листинг 2.4.1
`lstrcat`	Выполнить конкатенацию двух строк	Впервые упоминается в *главе 2.6*
`lstrcpy`	Скопировать одну строку в другую	Впервые упоминается в *главе 2.6*
`lstrlen`	Получить длину строки	Впервые упоминается в *главе 2.6*
`MapViewOfFile`	Скопировать файл или части файла в память	*Глава 3.5*, см. листинг 3.5.2
`MessageBox`	Выдать окно сообщения	*Глава 1.2*
`MoveFile`	Переместить файлы и каталоги	*Глава 2.8*
`MoveFileEx`	Переместить файлы и каталоги с настройкой операции	*Глава 2.8*
`MoveToEx`	Сменить текущую точку	*Глава 2.1*, см. листинг 2.1.6
`MoveWindow`	Установить новое положение окна	*Глава 3.1*, см. листинг 3.1.3

Таблица П1.1 *(продолжение)*

Название функции	Назначение функции	Где существенным образом упоминается
`OemToChar`	Функция перекодирует строку из кодировки для консольных окон в кодировку для графических окон	Глава 1.5
`OpenEvent`	Открыть событие	*Глава 3.2, см. разд. "События"*
`OpenProcess`	Открыть процесс	*Глава 3.5*, см. листинг 3.5.8
`OpenSCManager`	Открыть базу сервисов	*Глава 3.8*, см. листинги 3.8.2—3.8.4
`OpenSemaphor`	Открыть семафор	*Глава 3.2*, *см. разд. "Семафоры"*
`OpenService`	Открыть сервис	*Глава 3.8*, см. листинг 3.8.3
`PatBlt`	Заполнить заданную прямоугольную область	*Глава 2.1*, см. листинг 2.1.6
`Pie`	Рисовать сектор эллипса	Упоминается в *главе 2.1*
`PostMessage`	Аналогична `SendMessage`, но сразу возвращает управление	*Глава 3.3*, см. листинг 3.3.5
`PostQuitMessage`	Послать текущему приложению сообщение `WM_QUIT`	*Глава 1.2*
`ReadConsole`	Читать из консоли	*Глава 2.3*, см. листинг 2.3.2
`ReadFile`	Читать из файла или того, что было создано функцией `CreateFile`	*Глава 2.6*
`Rectangle`	Нарисовать прямоугольник	*Глава 2.1*, см. листинг 2.1.6
`recv`	Получить данные при взаимодействии программ посредством сокетов	*Глава 3.4*, см. листинги 3.4.5 и 3.4.5
`RegisterClass`	Зарегистрировать класс окон	*Глава 1.2*

Таблица П1.1 *(продолжение)*

Название функции	**Назначение функции**	**Где существенным образом упоминается**
`RegisterHotKey`	Зарегистрировать горячую клавишу	*Глава 2.5*, см. листинг 2.5.2
`RegisterServiceCtrlHandler`	С помощью этой функции регистрируется процедура-обработчик команд для данного логического сервиса	*Глава 3.8*, см. листинг 3.8.1
`ReleaseDC`	Удалить контекст при помощи `GetDC`	*Глава 2.1*, см. листинг 2.1.6
`ReleaseSemaphor`	Освободить семафор	*Глава 3.2*, *см. разд. "Семафоры"*
`RemoveDirectory`	Удалить каталог	Описание *см. в главе 2.8*
`ResetEvent`	Сбросить событие	*Глава 3.2*, *см. разд. "События"*
`ResumeThread`	Запустить "спящий" процесс	*Глава 3.2*, *см. разд. "Процессы и потоки"*
`RoundRect`	Нарисовать прямоугольник с округленными углами	*Глава 2.1*
`RtlMoveMemory`	Копировать блок памяти в другой блок. В справке по API-функциям она называется `MoveMemory`	*Глава 3.4*, см. листинг 3.4.3
`SelectObject`	Выбрать объект (перо, кисть) в указанном контексте	*Глава 2.1*, см. листинг 2.1.4
`Send`	Послать данные через сокет другой программе	*Глава 3.4*, см. листинги 3.4.5 и 3.4.6
`SendDlgItemMessage`	Послать сообщение управляющему элементу окна	*Глава 2.5*
`SendMessage`	Послать сообщение окну	*Глава 1.3*, см. листинг 1.3.2
`SetBkColor`	Установить цвет фона для вывода текста	*Глава 2.1*, см. листинг 2.1.1

Таблица П1.1 *(продолжение)*

Название функции	Назначение функции	Где существенным образом упоминается
`SetConsoleCursorPosition`	Установить курсор в заданную позицию в консоли	*Глава 2.3*, см. листинг 2.3.2
`SetConsoleScreenBufferSize`	Установить размер буфера консоли	*Глава 2.3*, см. листинг 2.3.2
`SetConsoleTextAttribute`	Установить цвет текста в консоли	*Глава 2.3*, см. листинг 2.3.2
`SetConsoleTitle`	Установить название окна консоли	*Глава 2.3*, см. листинг 2.3.2
`SetCurrentDirectory`	Установить текущий каталог	Подробно см. *главу 2.8*
`SetEvent`	Подать сигнал о наступлении события	*Глава 3.2*, *см. разд. "События"*
`SetFilePointer`	Установить указатель файла в заданное положение	*Глава 2.8*
`SetfileTime`	Установить временные характеристики файла	*Глава 2.6*
`SetFocus`	Установить фокус на заданное окно	*Глава 1.3*
`SetLocalTime`	Установить время и дату	Упоминается в *главе 3.1*
`SetMapMode`	Установить соотношение между логическими единицами и пикселами	Упоминается в *главе 2.1*
`SetMenu`	Назначить новое меню данному окну	*Глава 2.4*, см. листинг 2.4.2
`SetPixel`	Установить заданный цвет пиксела	*Глава 2.1*, см. листинг 2.1.6
`SetServiceStatus`	Данная функция устанавливает статус службы	*Глава 3.8*, см. листинг 3.8.1
`SetSystemTime`	Установить время, используя гринвичские координаты	Упоминается в *главе 3.1*
`SetTextColor`	Установить цвет текста	*Глава 2.1*, см. листинг 2.1.1

Таблица П1.1 *(продолжение)*

Название функции	**Назначение функции**	**Где существенным образом упоминается**
`SetTimer`	Установить таймер	*Глава 3.1*
`SetViewportExtEx`	Установить область вывода	*Глава 2.1*
`SetViewportOrgEx`	Установить начало области вывода	*Глава 2.1*
`SetWindowLong`	Изменить атрибут уже созданного окна	*Глава 3.5*, см. листинг 3.5.3
`SetWindowsHookEx`	Установить процедуру-фильтр	*Глава 3.6*, см. листинг 3.6.3
`Shell_NotifyIcon`	Посредством данной функции можно поместить пиктограмму приложения на системную панель	*Глава 3.5*, см. листинг 3.5.1
`SHFileOperation`	Осуществляет групповую операцию над файлами и каталогами	*Глава 3.5*, см. листинг 3.5.6
`SHGetDesktopFolder`	Выводит диалоговое окно для выбора каталогов и файлов	*Глава 3.5*
`ShowWindow`	Показать окно, установить статус показа	*Глава 1.2*
`Shutdown`	Закрыть связь через сокеты	*Глава 3.4*, см. листинги 3.4.5 и 3.4.6
`Sleep`	Вызывает задержку	*Глава 3.2*, см. рис. 3.2.1
`socket`	Создать сокет	*Глава 3.4*, см. листинги 3.4.5 и 3.4.6
`StartServiceCtrlDispatcher`	В задачу данной функции входит регистрация сервисов и запуск диспетчера управления сервисами	*Глава 3.8*, см. листинг 3.8.1
`SystemTimeToFileTime`	Преобразовать структуру `SYSTIME` в структуру файлового времени	*Глава 2.5*
`TerminateProcess`	Уничтожить процесс	*Глава 3.2*, см. листинг 3.2.1

Таблица П1.1 *(продолжение)*

Название функции	Назначение функции	Где существенным образом упоминается
`TerminateThread`	Удалить поток	*Глава 3.2*, см. листинг 3.2.1
`TextOut`	Вывести текст в окно	*Глава 2.1*
`timeKillEvent`	Удалить таймер	*Глава 2.3*, см. листинг 2.3.4
`timeSetEvent`	Установить таймер	*Глава 2.3*, см. листинг 2.3.4
`TranslateAccelerator`	Транслировать акселераторные клавиши в команду выбора пункта меню	*Глава 2.4*, см. листинг 2.4.3
`TranslateMessage`	Транслировать клавиатурные сообщения в ASCII-коды	*Глава 1.2*
`UnhookWindowsHookEx`	Снять процедуру-фильтр	*Глава 3.6*, см. листинг 3.6.3
`UnmapViewOfFile`	Сделать указатель на отображаемый файл недействительным	*Глава 3.5*, см. листинг 3.5.2
`UnRegisterHotKey`	Снять регистрацию горячей клавиши	*Глава 2.5*, см. листинг 2.5.2
`UpdateWindow`	Обновить рабочую область окна	*Глава 1.2*
`VirtualAlloc`	Зарезервировать блок виртуальной памяти или отобразить на него физическую память	*Глава 3.6*, *см. разд. "Виртуальная память"*
`VirtualFree`	Снять резервирование с блока виртуальной памяти или сделать блок виртуальной памяти неотображенным	*Глава 3.6*, *см. разд. "Виртуальная память"*
`WaitForSingleObject`	Ожидает одно из двух событий: определенный объект сигнализирует о своем состоянии, вышло время ожидания (`TimeOut`). Работает с такими объектами, как семафор, событие, взаимоисключение, процесс, консольный ввод и др.	*Глава 3.2*, *см. разд. "Семафоры"*

Таблица П1.1 *(окончание)*

Название функции	**Назначение функции**	**Где существенным образом упоминается**
`WaitNamedPipe`	Ожидать, когда освободится именованный канал связи	*Глава 2.8*, *см. разд. "Каналы передачи информации (pipes)"*
`WNetAddConnection2`	Осуществляет соединение с сетевым ресурсом локальной сети	Глава 3.4, см. листинг 3.4.2
`WnetCancelConnection2`	Отсоединить от ресурса локальной сети	*Глава 3.4*
`WnetCloseEnum`	Найти все ресурсы локальной сети данного уровня	*Глава 3.4*, см. листинг 3.4.3
`WnetGetConnection`	Получить информацию о данном соединении	*Глава 3.4*
`WnetOpenEnum`	Открыть поиск ресурсов в локальной сети	*Глава 3.4*, см. листинг 3.4.3
`WriteConsole`	Вывод в консоль	*Глава 2.3*, см. листинг 2.3.1
`WSAStartup`	Активизация библиотеки поддержки сокетов	*Глава 3.4*, см. листинг 3.4.5
`wsprintf`	Преобразовать последовательность параметров в строку	*Глава 2.3*, см. листинг 2.3.4

В третьей колонке табл. П1.2 указываются не вообще все места, где упоминается данное сообщение, а места, где разъясняется смысл этого сообщения либо просто упоминается, если нет другой информации.

Таблица П1.2. *Сообщения операционной системы Windows*

Сообщение системы	**Назначение**	**Где существенным образом упоминается**
`WM_ACTIVATE`	Посылается функции окна перед активизацией и деактивизацией этого окна	*Глава 2.5*
`WM_ACTIVATEAPP`	Посылается функции окна перед активизацией окна другого приложения	*Глава 2.5*

Таблица П1.2 *(продолжение)*

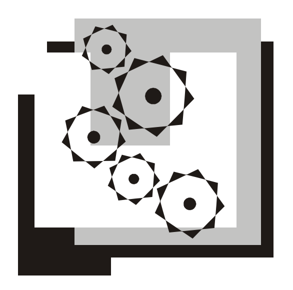
Сообщение системы	**Назначение**	**Где существенным образом упоминается**
WM_CHAR	Сообщение, возникающее при трансляции сообщения WM_KEYDOWN функцией TranslateMessage	*Главы 1.2, 2.5*
WM_CLOSE	Сообщение, приходящее в процедуру окна при его закрытии. Приходит до WM_DESTROY. Дальнейшее выполнение DefWindowProc, EndDialog или WindowsDestroy и вызывает появление сообщения WM_CREATE	*Глава 2.4*
WM_COMMAND	Сообщение, приходящее в функцию окна при наступлении события с управляющим элементом, пунктом меню, а также от акселератора	*Главы 1.3, 2.4*
WM_CREATE	Первое сообщение, приходящее в функцию окна при его создании. Приходит один раз	Подробнее *см. главу 1.2*
WM_DEADCHAR	Сообщение, возникающее при трансляции сообщения WM_KEYUP функцией TranslateMessage	*Глава 1.2*
WM_DESTROY	Сообщение, приходящее в функцию окна при его уничтожении	Подробнее *см. главу 1.2*
WM_GETTEXT	Посылается окну для получения текстовой строки, ассоциированной с данным окном (строка редактирования, заголовок окна и т. д.)	*Глава 1.3*, см. листинг 1.3.2
WM_HOTKEY	Генерируется при нажатии горячей клавиши	*Глава 2.5*, см. листинг 2.5.2
WM_INITDIALOG	Сообщение, приходящее в функцию диалогового окна вместо сообщения WM_CREATE	*Глава 2.4*
WM_KEYDOWN	Сообщение, генерируемое при нажатии клавиши и посылаемое окну, имеющему фокус ввода	*Главы 1.2, 2.4*
WM_KEYUP	Сообщение, генерируемое при отпускании клавиши и посылаемое окну, имеющему фокус ввода	*Главы 1.2, 2.4*
WM_LBUTTONDOWN	Сообщение генерируется при нажатии левой кнопки мыши	*Глава 1.2*

Таблица П1.2 *(продолжение)*

Сообщение системы	Назначение	Где существенным образом упоминается
`WM_MENUSELECT`	Посылается окну, содержащему меню, при выборе пункта меню	*Глава 2.5*
`WM_PAINT`	Сообщение посылается окну перед его перерисовкой	*Главы 1.2, 1.3*
`WM_QUIT`	Сообщение, приходящее приложению (не окну) при выполнении функции `PostQuitMessage`. При получении этого сообщения происходит выход из цикла ожидания и, как следствие, выход из программы	*Глава 1.2*, см. листинг 1.2.2
`WM_RBUTTONDOWN`	Сообщение генерируется при нажатии правой кнопки мыши	*Глава 1.2*, см. листинг 1.2.2
`WM_SETFOCUS`	Сообщение, посылаемое окну, после того, как оно получило фокус	*Глава 1.3*, см. листинг 1.3.2
`WM_SETICON`	Приложение посылает окну данное сообщение, чтобы ассоциировать с ним новую пиктограмму (значок)	*Глава 2.4*
`WM_SETTEXT`	Сообщение, используемое приложением для посылки текстовой строки окну и интерпретируемое в зависимости от типа окна (обычное окно — заголовок, кнопка — надпись на кнопке, окно редактирования — содержимое этого окна и т. д.)	*Глава 1.3*, см. листинг 1.3.2
`WM_SIZE`	Посылается функции окна после изменения его размера	*Глава 3.5*, см. листинг 3.5.1
`WM_SYSCHAR`	Сообщение, возникающее при трансляции сообщения `WM_SYSKEYDOWN` функцией `TranslateMessage`	*Глава 1.2*
`WM_SYSCOMMAND`	Генерируется при выборе пунктов системного меню или меню окна	*Глава 2.4*, см. также рис. 2.4.1
`WM_SYSDEADCHAR`	Сообщение, возникающее при трансляции сообщения `WM_SYSKEYUP` функцией `TranslateMessage`	*Глава 1.2*
`WM_SYSKEYDOWN`	Сообщение аналогично `WM_KEYDOWN`, но генерируется, когда нажата и удерживается еще и клавиша <Alt>	*Главы 1.2, 2.4*
`WM_SYSKEYUP`	Сообщение аналогично `WM_SYSDOWN`, но генерируется при отпускании клавиши	*Главы 1.2, 2.4*

Таблица П1.2 *(окончание)*

Сообщение системы	**Назначение**	**Где существенным образом упоминается**
WM_TIMER	Сообщение, приходящее в функцию окна или специально определенную процедуру после задания интервала таймера при помощи функции SetTimer	*Глава 3.1* полностью посвящена данному сообщению
WM_VKEYTOITEM	Сообщение окну приложения при нажатии какой-либо клавиши при наличии фокуса на данном списке. Список должен иметь свойство LBS_WANTKEYBOARDINPUT	*Глава 2.5*



Приложение 2

Справочник по командам и архитектуре микропроцессора Pentium

В литературе при описании команд микропроцессоров часто встречаются досадные ошибки. Стараясь избежать таких ошибок, автор выверял описание команд по нескольким источникам [3, 5, 6, 8—10]. Часть команд была проверена программным путем.

Регистры микропроцессора Pentium

Микропроцессор Pentium включает в себя регистры общего назначения, регистр флагов, сегментные регистры, управляющие регистры, системные адресные регистры, а также отладочные регистры. Особо следует отметить регистр `EIP`, который называют указателем команд. В нем всегда содержится адрес исполняемой команды относительно начала сегмента. К данному регистру нет прямого доступа, но косвенно многие команды изменяют его содержимое, например, команды передачи управления.

Регистры общего назначения

Перечислю их:

```
EAX = (16+AX=(AH+AL))
EBX = (16+BX=(BH+BL))
ECX = (16+CX=(CH+CL))
EDX = (16+DX=(DH+DL))
ESI = (16+SI)
EDI = (16+DI)
EBP = (16+BP)
ESP = (16+SP)
```

Регистры `EAX`, `EBX`, `EDX`, `ECX` называют рабочими регистрами. Регистры `EDI`, `ESI` — индексные регистры, играют особую роль в строковых операциях. Регистр `EBP`

обычно используется для адресации в стеке параметров и локальных переменных. Регистр `ESP` — указатель стека, автоматически модифицируется командами `PUSH`, `POP`, `RET`, `CALL`. Явно используется реже. Регистры `ESI`, `EDI`, `ESP`, `EBP` также имеют подрегистры. Например, первые 16 битов регистра `EDI` обозначаются как `DI`.

Регистр флагов

Содержит 32 бита. Вот используемые значения битов:

- 0-й бит — флаг переноса (`CF`), устанавливается в 1, если был перенос из старшего бита;
- 1-й бит — 1;
- 2-й бит — флаг четности, устанавливается в 1, если младший байт результата содержит четное количество единиц;
- 3-й бит — 0;
- 4-й бит — флаг вспомогательного переноса (`AF`), устанавливается в 1, если произошел перенос из третьего бита в четвертый;
- 5-й бит — 0;
- 6-й бит — флаг нуля (`ZF`), устанавливается в единицу, если результат операции — ноль;
- 7-й бит — флаг знака (`SF`), равен старшему биту результата;
- 8-й бит — флаг ловушки (`TF`), установка в единицу этого флага приводит к тому, что после каждой команды вызывается `INT 3`. Используется отладчиками в реальном режиме;
- 9-й бит — флаг прерываний (`IF`). Сброс этого флага в 0 приводит к тому, что микропроцессор перестает воспринимать прерывания;
- 10-й бит — флаг направления (`DF`). Данный флаг учитывается в строковых операциях. Если флаг равен 1, то в строковых операциях адрес автоматически уменьшается;
- 11-й бит — флаг переполнения (`OF`). Устанавливается в единицу, если результат операции над числом со знаком вышел за допустимые пределы;
- 12-й и 13-й биты — уровень привилегий ввода/вывода (`IOPL`). Определяет, какой привилегией должен обладать код, чтобы ему было разрешено выполнить команды ввода/вывода, а также другие привилегированные команды *(см. приложение 3)*;
- 14-бит — флаг вложенной задачи (`NT`);

- 15-й бит — 0;
- 16-й бит — флаг возобновления (RF). Используется совместно с регистрами точек отладочного останова;
- 17-й бит — в защищенном режиме включает виртуальный режим микропроцессора 8086 (VM);
- 18-й бит — флаг контроля выравнивания (AC). При равенстве этого флага 1 и при обращении к невыровненному операнду вызывает исключение 17;
- 19-й бит — виртуальная версия флага IF (VIF). Работает в защищенном режиме;
- 20-й бит — виртуальный запрос прерывания (VIP);
- 21-й бит — флаг доступности команды идентификации;
- биты 22—31 — должны быть сброшены в 0.

Сегментные регистры

CS — сегмент кода, DS — сегмент данных, SS — сегмент стека, ES, GS, FS — дополнительные регистры. Сегментные регистры являются 16-битными. Назначение сегментных регистров — участвовать в формировании адреса памяти либо напрямую, либо посредством селекторов, которые указывают на некоторую структуру (в дескрипторной таблице), определяющей сегмент, где находится формируемый адрес.

Управляющие регистры

Перечислю управляющие регистры.

- Регистр CR0.
  - 0-й бит — разрешение защиты (PE), переводит процессор в защищенный режим;
  - 1-й бит — мониторинг сопроцессора (MP), вызывает исключение 7 по каждой команде WAIT;
  - 2-й бит — эмуляция сопроцессора (EM), вызывает исключение 7 по каждой команде сопроцессора;
  - 3-й бит — бит переключения задач (TS). Позволяет определить, относится данный контекст сопроцессора к текущей задаче или нет. Вызывает исключение 7 при выполнении следующей команды сопроцессора;
  - 4-й бит — индикатор поддержки инструкций сопроцессора (ET);
  - 5-й бит — разрешение стандартного механизма сообщений об ошибке сопроцессора (NE);

- биты 6—15 — не используются;
- 16-й бит — разрешение защиты от записи на уровне привилегий супервизора (WP);
- 17-й бит — не используется;
- 18-й бит — разрешение контроля выравнивания (AM);
- биты 19—28 — не используются;
- 29-й бит — запрет сквозной записи кэша и циклов аннулирования (NW);
- 30-й бит — запрет заполнения кэша (CD);
- 31-й бит — включение механизма страничной переадресации.

❒ Регистр CR1 пока не используется.

❒ Регистр CR2 хранит 32-битный линейный адрес, по которому был получен последний отказ страницы памяти.

❒ Регистр CR3 — в старших 20 битах хранится физический базовый адрес таблицы каталога страниц.

Остальные используемые биты:

- 3-й бит — кэширование страниц со сквозной записью (PWT);
- 4-й бит — запрет кэширования страницы (PCD).

❒ Регистр CR4:

- 0-й бит — разрешение использования виртуального флага прерываний в виртуальном режиме микропроцессора 8086 (VME);
- 1-й бит — разрешение использования виртуального флага прерываний в защищенном режиме (PVI);
- 2-й бит — превращение инструкции RDTSC в привилегированную (TSD);
- 3-й бит — разрешение точек останова по обращению к портам ввода/вывода (DE);
- 4-й бит — включает режим адресации с 4-мегабайтными страницами (PSE);
- 5-й бит — включает 36-битное физическое адресное пространство (PAE);
- 6-й бит — разрешение исключения MC (MCE);
- 7-й бит — разрешение глобальной страницы (PGE);
- 8-й бит — разрешает выполнение команды RDPMC (PMC);
- 9-й бит — разрешает команды быстрого сохранения/восстановления состояния сопроцессора (FSR).

Системные адресные регистры

Эти регистры используются в защищенном режиме процессора Intel, в котором, в частности, и функционирует операционная система Windows.

Перечислю системные адресные регистры:

- GDTR — 6-байтный регистр, в котором содержится линейный адрес глобальной дескрипторной таблицы;
- IDTR — 6-байтный регистр, содержащий 32-битный линейный адрес таблицы дескрипторов обработчиков прерываний;
- LDTR — 10-байтный регистр, содержащий 16-битный селектор (индекс) для GDT и 8-байтный дескриптор;
- TR — 10-байтный регистр, содержащий 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий TSS текущей задачи.

Регистры отладки

К регистрам отладки относятся следующие регистры.

- DR0—DR3 — хранят 32-битные линейные адреса точек останова. Механизм работы регистров таков: любой формируемый программой адрес сравнивается с адресами, хранящимися в регистрах, и если есть совпадение, то генерируется исключение отладки (INT 1).
- DR6 (равносильно DR4) — отражает состояние контрольных точек. Биты этого регистра устанавливаются в соответствии с причинами, которые вызвали исключение отладки. Вот значащие биты этого регистра:
  - бит 0 — если значение этого бита равно нулю, то последнее исключение произошло по достижению контрольной точки, определенной в DR0;
  - бит 1 — аналогичен биту 0, но для регистра DR1;
  - бит 2 — аналогичен биту 0, но для регистра DR2;
  - бит 3 — аналогичен биту 0, но для регистра DR3;
  - бит 13 — служит для защиты регистров отладки;
  - бит 14 — если значение бита равно 1, то исключение произошло из-за того, что флаг ловушки (бит 8 в регистре флагов) равен 1;
  - бит 15 — если значение бита равно 1, то исключение вызвано переключением на задачу с установленным битом ловушки.
- DR7 (равносильно DR5) — управляет установкой контрольных точек. В этом регистре для каждого регистра отладки (DR0—DR3) имеются поля, определяющие условия, при которых следует сгенерировать прерывание. Первые

четыре пары битов регистра (8 битов), по паре на каждый регистр, задают, будет соответствующий регистр определять контрольную точку для локальной задачи (первый бит пары должен быть равен 1) или на все задачи системы (второй бит пары равен 1). Биты с 16 по 31 регистра определяют тип доступа, при котором будет срабатывать прерывание (при выборке команды, записи или чтении из памяти), и размер данных:

- биты 16—17, 20—21, 24—25, 28—29 определяют тип доступа: 00 — по команде, 01 — на запись, 11 — считывание и запись, 10 — не используется;
- биты 18—19, 22—23, 26—27, 30—31 задают размер операнда: 00 — байт, 01 — два байта, 11 — четыре байта, 10 — не используется.

Команды процессора

К основному набору я отношу все команды микропроцессора, кроме команд математического сопроцессора и команд MMX.

Принятые в табл. П2.1—П2.17 обозначения:

- `dest`, `src` — операнд-источник и операнд-получатель;
- `m` — обозначает операнд, расположенный в памяти;
- `r` — обозначает операнд — регистр процессора;
- `r8`, `r16`, `r32` — 8-,16-, 32-битные регистры процессора;
- `mm` — 64-битный регистр MMX;
- `m32` и `m64` — операнды, находящиеся в памяти и имеющие размер, соответственно, 32 и 64 бита;
- `ir32` — обычные регистры процессора;
- `imm` — непосредственный операнд (константа), размером в 1 байт.

Таблица П2.1. *Команды пересылки данных*

Команда	**Описание**
`MOV dest, src`	Пересылка данных в регистр из регистра, памяти или непосредственного операнда. Пересылка данных в память из регистра или непосредственного операнда. Например, `MOV AX,10`; `MOV EBX,ESI`; `MOV AL, BYTE PTR MEM`; `MOV DWORD PTR MEM,10000H`
`XCHG r/m, r`	Обмен данными между регистрами или регистром и памятью. Команда "память — память" в микропроцессоре Intel не предусмотрена

Таблица П2.1 *(продолжение)*

Команда	**Описание**
`BSWAP reg32`	Перестановка байтов из порядка "младший — старший" в порядок "старший — младший". Разряды 7—0 обмениваются с разрядами 31—24, а разряды 15—8 с разрядами 23—16. Команда появилась в 486-м микропроцессоре
`MOVSXB r, r/m`	Пересылка байта с его расширением до слова или двойного слова с дублированием знакового бита: `MOVSXB AX,BL`; `MOVSXB EAX, byte ptr mem`. Команда появилась, начиная с 386-го микропроцессора
`MOVSXW r, r/m`	Пересылка слова с расширением до двойного слова с дублированием знакового бита: `MOVSXW EAX,WORD PTR MEM`. Команда появилась, начиная с 386-го микропроцессора
`MOVZXB r, r/m`	Пересылка байта с его расширением до слова или двойного слова с дублированием нулевого бита: `MOVSXB AX,BL`; `MOVSXB EAX,byte ptr mem`. Команда появилась, начиная с 386-го микропроцессора
`MOVZXW r, r/m`	Пересылка слова с расширением до двойного слова с дублированием нулевого бита: `MOVZXW EAX,WORD PTR MEM`. Команда появилась, начиная с 386-го микропроцессора
`XLAT`	Загрузить в `AL` байт из таблицы в сегменте данных, на начало которой указывает `EBX` (`BX`), при этом начальное значение `AL` играет роль смещения
`LEA r, m`	Загрузка эффективного адреса. Например, `LEA EAX,MEM`; `LEA EAX,[EBX]`. Данная команда обладает магическими свойствами, позволяющими эффективно выполнять арифметические действия. Например, команда `LEA EAX,[EAX*8]` умножает содержимое `EAX` на 8, а команда `LEA EAX,[EAX][EAX*4]` — на 5. Команда `LEA ECX,[EAX][ESI+5]` эквивалентна трем (!) командам: `MOV ECX,EAX`/`ADD ECX,ESI`/`ADD ECX,5`
`LDS r, m`	Загрузить пару `DS:reg` из памяти. Причем вначале идет слово (или двойное слово), а в `DS` — последующее слово
`LES r, m`	Аналогично предыдущему, но для пары `ES:reg`
`LFS r, m`	Аналогично предыдущему, но для пары `FS:reg`
`LGS r, m`	Аналогично предыдущему, но для пары `GS:reg`
`LSS r, m`	Аналогично предыдущему, но для пары `SS:reg`

Таблица П2.1 *(продолжение)*

Команда	Описание
`SETcc r/m`	Набор команд. Проверяет условие "`cc`": если выполняется, то первый бит байта устанавливается в 1, в противном случае в 0. Условия аналогичны в условных переходах (`JE`, `JC`). Например, `SETE AL`. Команда появилась, начиная с 386-го микропроцессора. Полный перечень этих команд: • `SETA/SETNBE` — установить, если выше; • `SETAE/SETNB` — установить, если выше или равно; • `SETB/SETNAE` — установить, если ниже; • `SETBE/SETNA` — установить, если ниже; • `SETC` — установить, если перенос; • `SETE/SETZ` — установить, если ноль; • `SETG/SETNLE` — установить, если больше; • `SETGE/SETNL` — установить, если больше или равно; • `SETL/SETNGE` — установить, если меньше; • `SETLE/SETNG` — установить, если меньше или равно; • `SETNC` — установить, если нет переноса; • `SETNE/SETNZ` — установить, если меньше или равно; • `SETNO` — установить, если нет переполнения; • `SETNP/SETPO` — установить, если нет равенства; • `SETNS` — установить, если нет знака; • `SETO` — установить, если есть переполнения; • `SETP/SETPE` — установить, если есть равенство; • `SETS` — установить, если есть знак
`LAHF`	Загрузить флаги в `AH` (устарела)
`SAHF`	Сохранить `AH` в регистре флагов (устарела)
`CMOVX dest, src`	Набор команд условной пересылки: • `CMOVA/CMOVNBE` — переслать, если выше; • `CMOVAE/CMOVNB` — переслать, если выше или равно; • `CMOVB/CMOVNAE` — переслать, если ниже; • `CMOVBE/CMOVNA` — переслать, если ниже; • `CMOVC` — переслать, если перенос; • `CMOVE/CMOVZ` — переслать, если ноль; • `CMOVG/CMOVNLE` — переслать, если больше; • `CMOVGE/CMOVNL` — переслать, если больше или равно; • `CMOVL/CMOVNGE` — переслать, если меньше

Таблица П2.1 *(окончание)*

Команда	Описание
`CMOVX dest, src`	Набор команд условной пересылки: • `CMOVLE/CMOVNG` — переслать, если меньше или равно; • `CMOVNC` — переслать, если нет переноса; • `CMOVNE/CMOVNZ` — переслать, если меньше или равно; • `CMOVNO` — переслать, если нет переполнения; • `CMOVNP/CMOVPO` — переслать, если нет равенства; • `CMOVNS` — переслать, если нет знака; • `CMOVO` — переслать, если есть переполнения; • `CMOVP/CMOVPE` — переслать, если есть равенство; • `CMOVS` — переслать, если есть знак

Таблица П2.2. *Команды ввода/вывода*

Команда	Описание
`IN AL(AX,EAX), Port` `IN AL(AX,EAX), DX`	Ввод в аккумулятор из порта ввода/вывода. Порт адресуется непосредственно или через регистр `DX`
`OUT port, AL(AX,EAX)` `OUT DX, AL(AX,EAX)`	Вывод в порт ввода/вывода. Порт адресуется непосредственно или через регистр `DX`
`[REP] INSB` `[REP] INSW` `[REP] INSD`	Выводит данные из порта, адресуемого регистром `DX` в ячейку памяти `ES:[EDI/DI]`. После ввода байта, слова или двойного слова производится коррекция `EDI/DI` на 1, 2 или 4. При наличии префикса `REP`-процесс продолжается, пока содержимое `CX` не станет равным 0
`[REP] OUTSB` `[REP] OUTSW` `[REP] OUTSD`	Выводит данные из ячейки памяти, определяемой регистрами `DS:[ESI/SI]`, в выходной порт, адрес которого находится в регистре `DX`. После вывода байта, слова, двойного слова производится коррекция указателя `ESI/SI` на 1, 2, 4

Таблица П2.3. *Инструкции работы со стеком*

Команда	Описание
`PUSH r/m`	Поместить в стек слово или двойное слово. Поскольку при включении в стек слова нарушается выравнивание стека по границам двойных слов, рекомендуется в любом случае помещать в стек двойное слово
`PUSH const`	Поместить в стек непосредственный 32-битный операнд

Таблица П2.3 (окончание)

Команда	Описание
PUSHA	Поместить в стек 16-битные регистры AX, BX, CX, DX, SI, DI, BP, SP. Команда появилась, начиная с 80186-го процессора
POP reg/mem	Извлечь из стека слово или двойное слово
POPA	Извлечение из стека данных в 16-битные регистры AX, BX, CX, DX, SI, DI, BP, SP. Команда появилась, начиная с 80186-го процессора
PUSHAD	Поместить в стек 32-битные регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. Команда появилась, начиная с 386-го процессора
POPAD	Извлечение из стека данных в 32-битные регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. Команда появилась, начиная с 386-го процессора
PUSHF	Помещение в стек регистра флагов
POPF	Извлечь данные из регистра флагов

Таблица П2.4. *Инструкции целочисленной арифметики*

Команда	Описание
ADD dest, src	Сложение двух операндов. Первый операнд может быть регистром или ячейкой памяти, второй — регистром, ячейкой памяти, константой. Операция невозможна, когда оба операнда являются ячейками памяти
XADD dest, src	Данная операция производит вначале обмен операндами, а затем выполняет операцию ADD. Команда введена, начиная с 486-го процессора
ADC dest, src	Сложение с учетом флага переноса — в младший бит добавляется бит (флаг) переноса
INC r/m	Инкремент операнда
SUB dest, src	Вычитание двух операндов. Остальное аналогично сложению (команда ADD)
SBB dest, src	Вычитание с учетом бита переноса. Из младшего бита вычитается бит (флаг) переноса
DEC r/m	Декремент операнда
CMP r/m, r/m	Вычитание без изменения операндов (сравнение)

Таблица П2.4 *(продолжение)*

Команда	Описание
CMPXCHG r, m, a	Сравнение с обменом. Воспринимает три операнда (регистр — операнд-источник, ячейка памяти — операнд-получатель, аккумулятор, т. е. AL, AX или EAX). Если значения в операнде-получателе и аккумуляторе равны, операнд-получатель заменяется операндом-источником, исходное значение операнда-получателя загружается в аккумулятор. Команда появилась, начиная с 486-го процессора
CMPXCHG8B r, m ,a	Сравнение и обмен восемью байтами. Команда появилась, начиная с Pentium. Сравнивается число, находящееся в паре регистров EDX:EAX с восьмибайтным числом в памяти
NEG r/m	Изменение знака операнда
AAA	Коррекция после ASCII-сложения. Коррекция результата двоичного сложения двух неупакованных двоично-десятичных чисел. Например, AX содержит число 9H. Пара команд ADD AL,8/AAA приводит к тому, что в AX будет содержаться 0107, т. е. ASCII-число 17
AAS	Коррекция после ASCII-вычитания. Коррекция результата двоичного вычитания двух неупакованных двоично-десятичных чисел. Например: MOV AX,205H ; загрузить ASCII-число 25 SUB AL,8 ; двоичное вычитание AAS В результате AX содержит код 107H, т. е. неупакованное двоично-десятичное число 17
AAM	Коррекция после ASCII-умножения. Для этой команды предполагается, что в регистре AX находится результат двоичного умножения двух десятичных цифр (диапазон от 0 до 81). После выполнения команды образуется двухбайтное произведение в регистре AX в ASII-формате
AAD	Коррекция перед ASCII-делением. Предполагается, что младшая цифра находится в AL, а старшая — в AH
DAA	Коррекция после BCD-сложения[1]
DAS	Коррекция после BCD-вычитания

[1] Напоминаю, что ASCII-число предполагает одну цифру на один байт, BCD-число — одну цифру на половину байта. Таким образом, скажем, в регистре AX может находиться двухразрядное ASCII-число и четырехразрядное BCD-число.

Таблица П2.4 *(окончание)*

Команда	Описание
MUL r/m	Умножение AL (AX, EAX) на целое беззнаковое число. Результат, соответственно, будет содержаться в AX, DX:AX, EDX:EAX
IMUL r/m	Знаковое умножение (аналогично MUL). Все операнды считаются знаковыми. Команда IMUL имеет также двухоперандный и трехоперандный вид. Двухоперандный вид: IMUL r,src r<-r*src Трехоперандный вид: IMUL dst, src, imm dst<-src*imm
DIV r/m (src)	Беззнаковое деление. Аналогично беззнаковому умножению. Осуществляет деление аккумулятора и его расширения (AH:AL, DX:AX, EDX:EAX) на делитель src. Частное помещается в аккумулятор, а остаток — в расширение аккумулятора
IDIV r/m	Знаковое деление. Аналогично беззнаковому
CBW	Расширение байта (AL) до слова с копированием знакового бита
CWD	Расширение слова (AX) до двойного слова (DX:AX) с копированием знакового бита
CWDE	Расширение слова (AX) до двойного слова (EAX) с копированием знакового бита
CDQ	Преобразование двойного слова (EAX) в учетверенное слово (EDX:EAX)

Таблица П2.5. *Логические операции*

Команда	Описание
AND dest, src	Логическая операция "И". Обнуление битов dest, которые равны нулю в src
TEST dest, src	Аналогична "И", но не меняет dest. Используется для проверки ненулевых битов
OR dest, src	Логическая операция "ИЛИ". В dest устанавливаются биты, отличные от нуля в src
XOR dest, src	Исключающее "ИЛИ"
NOT dest	Переключение всех битов (инверсия)

Таблица П2.6. *Сдвиговые операции*

Команда	**Описание**
`RCL/RCR dest, src`	Циклический сдвиг влево/вправо через бит переноса `CF`. `src` может быть либо `CL`, либо непосредственным операндом
`ROL/ROR dest, src`	Аналогична командам `RCL/RCR`, но иначе работает с флагом `CF`. Флаг `CF` не участвует в цикле сдвига, но в него попадает бит, перешедший с начала на конец или наоборот
`SAL/SAR dest, src`	Сдвиг влево/вправо. Называется еще арифметическим сдвигом. При сдвиге вправо дублируется старший бит. При сдвиге влево младший бит заполняется нулем. "Вытолкнутый" бит помещается в `CF`
`SHL/SHR dest, src`	Логический сдвиг влево/вправо. Сдвиг вправо отличается от `SAR` тем, что и старший бит заполняется нулем
`SHLD/SHRD dest, src, count`	Трехоперандные команды сдвига влево/вправо. Первым операндом, как обычно, может быть либо регистр, либо ячейка памяти, вторым операндом должен быть регистр общего назначения, третьим — регистр `CL` или непосредственный операнд. Суть операции заключается в том, что `dest` и `src` вначале объединяются, а потом производится сдвиг на количество бит `count`. Результат снова помещается в `dest`

Замечание

Начиная с 386-го микропроцессора, непосредственный операнд `src` в сдвиговых операциях может быть не только 1, но произвольным числом. В ранних версиях для количества сдвигов использовался регистр `CL`.

Таблица П2.7. *Строковые операции*

Команда	**Описание**
`REP`	Префикс, означающий повтор строковой операции до обнуления `ECX`. Префикс имеет также разновидности `REPZ` (`REPE`) — выполнять, пока не ноль (`ZF=1`), `REPNZ` (`REPNE`) — выполнять, пока ноль
`MOVS dest, src`	Команда передает байт, слово или двойное слово из цепочки, адресуемой `DS:[ESI]`, в цепочку `dest`, адресуемую `ES[EDI]`. При этом `EDI` и `ESI` автоматически корректируются согласно значению флага `DF`. Допускается явная спецификация `MOVSB` (`byte`) — побайтовое копирование, `MOVSW` (`word`) — копирование словами, `MOVSD` (`word`) — четырехбайтовое копирование. `dest` и `src` можно явно не указывать

Таблица П2.7 (окончание)

Команда	Описание
`LODS src`	Команда загрузки цепочки в аккумулятор. Имеет разновидности `LODSB`, `LODSW`, `LODSD`. При выполнении команды байт, слово, двойное слово загружаются, соответственно, в `AL`, `AX`, `EAX`. При этом `ESI` автоматически изменяется на 1 в зависимости от значения флага `DF`. Префикс `REP` не используется
`STOS dest`	Команда, обратная `LODS`, т. е. передает байт, слово или двойное слово из аккумулятора в цепочку и автоматически корректирует `EDI`
`SCAS dest`	Команда сканирования цепочки. Команда вычитает элемент цепочки `dest` из содержимого аккумулятора (`AL`/`AX`/`EAX`) и модифицирует флаги. Префикс `REPNE` позволяет найти в цепочке нужный элемент
`CMPS dest, src`	Команда сравнения цепочек. Данная команда производит вычитание байта, слова или двойного слова цепочки `dest` из соответствующего элемента цепочки `src`. В зависимости от результата вычитания модифицируются флаги. Регистры `EDI` и `ESI` автоматически продвигаются на следующий элемент. При использовании префикса `REPE` команда означает — сравнивать, пока не будет достигнут конец цепочки или пока элементы не будут равны. При использовании префикса `REPNE` команда означает — сравнивать, пока не достигнут конец цепочки или пока элементы будут равны

Таблица П2.8. *Команды управления флагами*

Команда	Описание
`CLC`	Сброс флага переноса
`CMC`	Инверсия флага переноса
`STC`	Установка флага переноса
`CLD`	Сброс флага направления
`STD`	Установка флага направления
`CLI`	Запрет маскируемых аппаратных прерываний
`STI`	Разрешение маскируемых аппаратных прерываний
`CTS`	Сброс флага переключения задач

Таблица П2.9. *Команды передачи управления*

Команда	**Описание**
`JMP target`	Имеет пять форм, различающихся расстоянием назначения от текущего адреса и способом задания целевого адреса. При работе в Windows используется в основном внутрисегментный переход (`NEAR`) в пределах 32-битного сегмента. Адрес перехода может задаваться непосредственно (в программе это метка) или косвенно, т. е. содержаться в ячейке памяти или регистре (`JMP [EAX]`). Другой тип перехода — короткий переход (`SHORT`), занимает всего 2 байта. Диапазон смещения, в пределах которого происходит переход: –128 — 127. Использование такого перехода весьма ограниченно. Межсегментный переход может иметь следующий вид: `JMP FWORD PTR L`, где `L` — указатель на структуру, содержащую 48-битный адрес, в начале которого 32-битный адрес смещения, затем 16-битный селектор (сегмента, шлюза вызова, сегмента состояния задачи). Возможен также и такой вид межсегментного перехода: `JMP FWORD ES:[EDI]`
Условные переходы	• `JA/JNBE` — перейти, если выше. • `JAE/JNB` — перейти, если выше или равно. • `JB/JNAE` — перейти, если ниже. • `JBE/JNA` — перейти, если ниже. • `JC` — перейти, если перенос. • `JE/JZ` — перейти, если ноль. • `JG/JNLE` — перейти, если больше. • `JGE/JNL` — перейти, если больше или равно. • `JL/JNGE` — перейти, если меньше. • `JLE/JNG` — перейти, если меньше или равно. • `JNC` — перейти, если нет переноса. • `JNE/JNZ` — перейти, если меньше или равно. • `JNO` — перейти, если нет переполнения. • `JNP/JPO` — перейти, если нет равенства. • `JNS` — перейти, если нет знака. • `JO` — перейти, если есть переполнение. • `JP/JPE` — перейти, если есть равенство. • `JS` — перейти, если есть знак. • `JCXZ` — переход, если `CX=0`. • `JECXZ` — переход, если `ECX=0`. В плоской модели команды условного перехода осуществляют переход в пределах 32-битного регистра

Таблица П2.9 *(окончание)*

Команда	**Описание**
Команды управления циклом. Все команды этой группы уменьшают содержимое регистра `ECX`	`LOOP` — переход, если содержимое `ECX` не равно нулю. `LOOPE` (`LOOPZ`) — переход, если содержимое `ECX` не равно нулю и флаг `ZF=1`. `LOOPNE` (`LOOPNZ`) — переход, если содержимое `ECX` не равно нулю и флаг `ZF=0`
`CALL target`	Передает управление процедуре (метке), следующей за `CALL`-командой, с сохранением в стеке адреса. В плоской модели адрес возврата представляет собой 32-битное смещение. Межсегментный вызов предполагает сохранение в стеке селектора и смещения, т. е. 48-битной величины (16 битов — селектор и 32 бита — смещение)
`RET [N]`	Возврат из процедуры. Необязательный параметр `N` предполагает, что команда также автоматически чистит стек (освобождает `N` байтов). Команда имеет разновидности, которые выбираются ассемблером автоматически, в зависимости от того, является процедура ближней или дальней. Можно, однако, и явно указать тип возврата (`RETN` или `RETF`). В случае плоской модели по умолчанию берется `RETN` с четырехбайтным адресом возврата

Таблица П2.10. *Команды поддержки языков высокого уровня*

Команда	**Описание**
`ENTER par1, par2`	Подготовка стека при входе в процедуру *(см. главу 1.2)*
`LEAVE`	Приведение стека в исходное состояние
`BOUND REG16, MEM16` или `BOUND REG32, MEM32`	Предполагается, что регистр содержит текущий индекс массива, а второй операнд определяет в памяти два слова или два двойных слова. Первое считается минимальным значением индекса, а второе — максимальным. Если текущий индекс оказывается вне границ, то генерируется команда `INT 5`. Используется для контроля нахождения индекса в заданных рамках, что является важным средством отладки

Таблица П2.11. *Команды прерываний*

Команда	**Описание**
`INT n`	Двухбайтная команда. Вначале в стек помещается содержимое регистра флагов, затем полный адрес возврата. Кроме того, сбрасывается флаг `TF`. После этого осуществляется косвенный переход через `n`-й элемент дескрипторной таблицы прерываний. Однобайтная команда `INT 3` называется прерыванием контрольного останова и используется в программах-отладчиках

Таблица П2.11 *(окончание)*

Команда	Описание
INTO	Равносильна команде INT 4, если флаг переполнения OF=1, если OF=0 — команда не производит никакого действия
IRET	Команда возврата из прерываний. Извлекает из стека сохраненные в нем адрес возврата и регистр флагов. Бит уровня привилегий будет модифицироваться только в том случае, если текущий уровень привилегий равен 0

Таблица П2.12. *Команды синхронизации процессора*

Команда	Описание
HLT	Останавливает процессор. Из такого останова процессор может быть выведен внешним прерыванием
LOCK	Представляет собой префикс блокировки шины. Он заставляет процессор сформировать сигнал LOCK# на время выполнения находящейся за префиксом команды. Этот сигнал блокирует запросы шины другими процессорами в мультипроцессорной системе
NOP	Холостая команда. Не производит никаких действий. Используется для удаления ненужных команд в исполняемом модуле, без изменения его длины
WAIT (FWAIT)	Синхронизация с сопроцессором. Большинство команд сопроцессора автоматически вырабатывают эту команду

Таблица П2.13. *Команды обработки цепочки бит*[2]

Команда	Описание
BSF(BSR) dest, src	dest — 16- или 32-битный регистр. src — регистр или ячейка памяти. При выполнении команды BSF операнд src просматривается с младших, а в команде BSR — со старших битов. Номер первого встречного бита, находящегося в состоянии 1, помещается в регистр dest, флажок ZF сбрасывается в 0. Если src содержит 0, то ZF=1, а содержимое dest не определено
BT dest, src	Тестирование бита с номером из src в dest и перенос его во флаг CF
BTC dest, src	Проверка и инвертирование бита из src в dest
BTR dest, src	Проверка и сброс бита из src в dest
BTS dest, src	Проверка и установка бита из src в dest

[2] Эти команды появились в 386-м процессоре.

Таблица П2.14. *Команды управления защитой*

Команда	Описание
`LGDT src`	Загрузка GDTR из памяти. `src` указывает на 6-байтную величину
`SGDT dest`	Сохранить GDTR в памяти
`LIDT src`	Загрузить IDTR из памяти
`SIDT dest`	Сохранить IDTR в памяти
`LLDT src`	Загрузить LDTR из памяти (16 битов)
`SLDT dest`	Сохранить LDTR в регистре или памяти (16 битов)
`LMSW src`	Загрузка MSW
`SMSW dest`	Сохранить MSW в регистре или памяти (16 битов)
`LTR src`	Загрузка регистра задачи из регистра или памяти (16 битов)
`STR dest`	Сохранение регистра задачи в регистре или памяти (16 битов)
`LAR dest, src`	Загрузка старшего байта `dest` байтом прав доступа дескриптора `src`
`LSL dest, src`	Загрузка `dest` пределом сегмента, дескриптор которого задан `src`
`ARPL r/m, r`	Выравнивание `RPL` в селекторе до наибольшего числа из текущего уровня и заданного операндом
`VERR seg`	Верификация чтения: установка `ZF=1`, если задаче позволено чтение в сегменте `SEG`
`VERW seg`	Верификация записи: установка `ZF=1`, если задаче позволена запись в сегменте `SEG`

Таблица П2.15. *Команды обмена с управляющими регистрами*

Команда	Описание
`MOV CRn, src`	Загрузка управляющего регистра `CRn`
`MOV dest, CRn`	Чтение управляющего регистра `CRn`
`MOV DRn, src`	Загрузка регистра отладки `DRn`
`MOV dest, DRn`	Чтение регистра отладки `DRn`
`MOV TRn, src`	Загрузка регистра тестирования `TRn`

Таблица П2.15 *(окончание)*

Команда	Описание
`MOV dest, TRn`	Чтение регистра тестирования `TRn`
`RDTSC`	Чтение счетчика тактов. Значение счетчика тактов помещается в пару регистров `EDX:EAX`

Таблица П2.16. *Команды идентификации и управления архитектурой*

Команда	Описание
`CPUID`	Получение информации о процессоре. Требует параметр в регистре `EAX`. Если `EAX=0`, процессор возвращает символьную строку, специфичную для производителя, в регистрах `EBX`, `EDX`, `ECX`. Процессоры AMD возвращают строку "AuthenticAMD", процессоры Intel — "GenuineIntel". Если `EAX=1`, в младшем слове регистра `EAX` возвращается код идентификации. Если `EAX=2`, в регистрах `EAX`, `EBX`, `ECX`, `EDX` возвращаются параметры конфигурации процессора
`RDMSR r/m`	Чтение модельно-специфического регистра в `ECX`
`WRMSR r/m`	Запись `ECX` в модельно-специфический регистр
`SYSENTER`	Системный вызов
`SYSEXIT`	Возврат из системного вызова

Таблица П2.17. *Команды управления кэшированием*

Команда	Описание
`INVD`	Аннулирование данных в первичном кэше без обратной записи
`WBINVD`	Обратная запись модифицированных строк и аннулирование кэш-памяти
`INVLPG r/m`	Аннулирование элемента таблицы трансляции TLB (TLB — буфер ассоциативной трансляции таблиц каталогов и страниц памяти)

Замечание

Внутренний кэш появился в процессоре, начиная с 486-го. Процессоры 486 и Pentium имеют внутренний кэш первого уровня, Pentium Pro и Pentium II имеют уже и вторичный кэш.

Команды арифметического сопроцессора

Описание работы арифметического сопроцессора см. в [1, 5]. Здесь мы коснемся основных положений работы арифметического сопроцессора.[3]

- Арифметический сопроцессор работает со своим набором команд и своим набором регистров. Однако выборку команд сопроцессора осуществляет процессор.
- Арифметический сопроцессор выполняет операции со следующими типами данных: целое слово (16 битов), короткое целое (32 бита), длинное слово (64 бита), упакованное десятичное число (80 битов), короткое вещественное число (32 бита), длинное вещественное число (64 бита), расширенное вещественное число (80 битов).
- При выполнении операции сопроцессором процессор ждет завершения этой операции. Другими словами, перед каждой командой сопроцессора ассемблером автоматически генерируется команда, проверяющая, занят сопроцессор или нет. Если сопроцессор занят, процессор переводится в состояние ожидания. Иногда программисту требуется вручную ставить команду ожидания (`WAIT`) после команды сопроцессора.
- Сопроцессор имеет восемь 80-битных рабочих регистров, представляющих собой стековую кольцевую структуру. Регистры называются `R0`, `R1`, ..., `R7`, но доступ к ним напрямую невозможен. Каждый регистр может занимать любое положение в стеке. Название стековых (относительных) регистров — `ST(0)`, `ST(1)`, `ST(2)`, `ST(3)`, `ST(4)`, `ST(5)`, `ST(6)`, `ST(7)`. Кроме того, имеется еще регистр состояния, по флагам которого можно, в частности, судить о результате выполненной операции. Регистр управления содержит в себе биты, влияющие на выполнение команд сопроцессора.
- Регистр тегов содержит 16 битов, описывающих содержание регистров сопроцессора: по два бита на каждый рабочий регистр. Тег говорит о содержимом регистре данных. Вот значение тегов: 00 — действительное ненулевое число, 01 — истинный ноль, 10 — специальные числа, 11 — отсутствие данных.
- При вычислении с помощью команд сопроцессора большую роль играют исключения или особые ситуации. Типичной особой ситуацией является деление на 0. Биты особых ситуаций хранятся в регистре состояний. Учет особых ситуаций необходим для получения правильных результатов.

---

[3] Мы пользуемся несколько устаревшим названием. Правильнее было бы это назвать числовым процессором.

Специальные случаи:

- положительный ноль (все биты нули);
- отрицательный ноль (знаковый бит равен 1);
- положительная бесконечность (знаковый бит 0, все биты мантиссы 0, все биты экспоненты 1);
- отрицательная бесконечность (знаковый бит 1, все биты мантиссы 0, все биты экспоненты 1);
- денормализованное число (все биты экспоненты 0);
- неопределенное число (знаковый бит 1, все биты экспоненты 1, первый бит мантиссы, а для 80-битного числа два бита 1, остальные 0);
- нечисловой экземпляр SNAN (все биты экспоненты 1, первый бит мантиссы 0, а для 80-битного числа первые два бита 10, а среди остальных битов есть 1);
- нечисловой экземпляр QNAN (все биты экспоненты 1, первый бит мантиссы 0, а для 80-битного числа два первых равны нулю, среди остальных битов мантиссы есть 1);
- неподдерживаемое число (ситуации, не соответствующие стандартным числам и не описанные в специальных случаях).

К особым ситуациям относятся следующие:

- неточный результат (округление);
- недействительная операция;
- деление на ноль;
- антипереполнение (слишком маленький результат);
- переполнение (слишком большой результат);
- денормализованный операнд.

❒ Регистр (слово) состояния:

- 0-й бит — флаг недопустимой операции;
- 1-й бит — флаг денормализованной операции;
- 2-й бит — флаг деления на ноль;
- 3-й бит — флаг переполнения;
- 4-й бит — флаг антипереполнения;
- 5-й бит — флаг неточного результата;
- 6-й бит — ошибка стека;

- 7-й бит — общий флаг ошибки;
- биты 8—10, 14 — флаги условий;
- биты 11—13 — число, показывающее, какой регистр является вершиной;
- 15-й бит — флаг занятости.

❒ Регистр (слово) управления:

- 0-й бит — маска недействительной операции;
- 1-й бит — маска денормализованного операнда;
- 2-й бит — маска деления на ноль;
- 3-й бит — маска переполнения;
- 4-й бит — маска антипереполнения;
- 5-й бит — маска неточного результата;
- 6-й и 7-й биты — резерв;
- 8-й и 9-й биты — управление точностью;
- 10-й и 11-й биты — управление округлением;
- 12-й бит — управление бесконечностью;
- биты 13—15 — резерв.

В таблицах П2.18—П2.22 дан перечень инструкций арифметического сопроцессора.

Таблица П2.18. *Команды передачи данных*

Команда	**Описание**
`FLD src`	Загрузить вещественное число в `ST(0)` (вершину стека) из области памяти. Область памяти может быть 32-, 64-, 80-битной
`FILD src`	Загрузить целое число в `ST(0)` из памяти. Область памяти может быть 16-, 32-, 64-битной
`FBLD src`	Загрузить BCD-число в `ST(0)` из 80-битной области памяти
`FLDZ`	Загрузить 0 в `ST(0)`
`FLD1`	Загрузить 1 в `ST(0)`
`FLDPI`	Загрузить `PI` в `ST(0)`
`FLDL2T`	Загрузить `LOG2(10)` в `ST(0)`
`FLDTL2E`	Загрузить `LOG2(e)` в `ST(0)`
`FLDLG2`	Загрузить `LG(2)` в `ST(0)`

Таблица П2.18 *(окончание)*

Команда	Описание
FLDLN2	Загрузить LN(2) в ST(0)
FST dest	Запись вещественного числа из ST(0) в память. Область памяти 32-, 64- или 80-битная
FSTP dest	Запись вещественного числа из ST(0) в память. Область памяти 32-, 64- или 80-битная. При этом происходит выталкивание вершины из стека
FBST dest	Запись BCD-числа в память. Область памяти 80-битная
FBSTP dest	Запись BCD-числа в память. Область памяти 80-битная. При этом происходит выталкивание вершины из стека
FXCH ST(i)	Обмен значениями вершины стека и регистра i
FCMOVc dest, src	Команда условной пересылки данных. Копирование ST(i) (src) в ST(0) (dest). Команда может иметь следующий вид: • FCMOVE — копировать, если равно (ZF=1); • FCMOVNE — копировать, если не равно (ZF=0); • FCMOVB — копировать, если меньше (CF=1); • FCMOVBE — копировать, если меньше или равно (CF=1 и ZF=1); • FCMOVNB — копировать, если меньше (CF=0); • FCMOVNBE — копировать, если меньше или равно (CF=0 и ZF=1); • FCMOVU — копировать, если не сравнимы (PF=1); • FCMOVNU — копировать, если сравнимы (PF=0)

Таблица П2.19. *Команды сравнения данных*

Команда	Описание
FCOM	Сравнение вещественных чисел ST(0) и ST(1). Флаги устанавливаются, как при операции вычитания ST(0)-ST(1)
FCOM src	Сравнение ST(0) с операндом в памяти. Операнд может быть 32- или 64-битным
FCOMP src	Сравнение вещественного числа в ST(0) с операндом с выталкиванием ST(0) из стека. Операнд может быть регистром и областью памяти

Таблица П2.19 (окончание)

Команда	Описание
`FCOMPP`	Сравнение `ST(0)` и `ST(1)` с двойным выталкиванием из стека
`FICOM src`	Сравнение целых чисел в `ST(0)` с операндом. Операнд может быть 16- или 32-битным
`FICOMP src`	Сравнение целых чисел в `ST(0)` с операндом. Операнд может быть 16- или 32-битной областью памяти или регистром. При выполнении операции происходит выталкивание `ST(0)` из стека
`FTST`	Проверка `ST(0)` на ноль
`FUCOM ST(i)`	Сравнение `ST(0)` с `ST(i)` без учета порядков
`FUCOMP ST(i)`	Сравнение `ST(0)` с `ST(i)` без учета порядков. При выполнении операции происходит выталкивание из стека
`FUCOMPP ST(i)`	Сравнение `ST(0)` с `ST(i)` без учета порядков. При выполнении операции происходит двойное выталкивание из стека
`FXAM`	Анализ содержимого вершины стека. Результат помещается в биты `C3`, `C2`, `C0`: • 000 — неподдерживаемый формат; • 001 — не число; • 010 — нормализованное число; • 011 — бесконечность; • 100 — ноль; • 101 — пустой операнд; • 110 — денормализованное число
`FCOMI src`	Сравнить и установить флаги. Команда `FCOMI` и следующие за ней команды `FCOMIP`, `FUCOMI`, `FUCOMIP` воздействуют на биты регистра флагов так: • `ST(0)` > `src` `ZF=0`, `PF=0`, `CF=0`; • `ST(0)` < `src` `ZF=0`, `PF=0`, `CF=1`; • `ST(0)` = `src` `ZF=1`, `PF=0`, `CF=0`. Если операнды несравнимы, то все три флага равны 1
`FCOMIP src`	Сравнить, установить биты и вытолкнуть
`FUCOMI src`	Сравнить без учета порядков и установить флаги
`FUCOMIP src`	Сравнить без учета порядков, установить флаги и вытолкнуть

Таблица П2.20. Арифметические команды

Команда	Описание
FADD src FADD ST(i), ST	Сложение вещественных чисел. ST(0)<-ST(0)+src, где src — 32- или 64-битное число ST(i)<-ST(i)+ST(0)
FADDP ST(i), ST	Сложение вещественных чисел, ST(i)<-ST(i)+ST(0). При выполнении операции происходит выталкивание из стека
FIADD src	Сложение целых чисел. ST(0)<-ST(0)+src, src — 16- или 32-битное число
FSUB src FSUB ST(i), ST	Вычитание вещественных чисел. ST(0)<-ST(0)-src, где src — 32- или 64-битное число. ST(i)<-ST(i)-ST(0)
FSUBP ST(i), ST	Вычитание вещественных чисел, ST(i)<-ST(i)-ST(0). При выполнении операции происходит выталкивание из стека
FSUBR ST(i), ST	Обратное вычитание вещественных чисел. ST(0)<-ST(i)-ST(0)
FSUBRP ST(i), ST	Обратное вычитание вещественных чисел. ST(0)<-ST(i)-ST(0). При выполнении операции происходит выталкивание из стека
FISUB src	Вычитание целых чисел. ST(0)<-ST(0)-src, где src — 16- или 32-битное число
FISUBR src	Вычитание целых чисел. ST(0)<-ST(0)-src, где src — 16- или 32-битное число. При выполнении операции происходит выталкивание из стека
FMUL FMUL ST(i) FMUL ST(i), ST	Умножение двух операндов. В первом случае ST(0)<-ST(0)*ST(1). Во втором случае ST(0)<-ST(i)*ST(0). В третьем случае ST(i)<-ST(i)*ST(0)
FMULP ST(i), ST(0)	Умножение и выталкивание из стека. ST(i)<-ST(i)*ST(0)
FIMUL src	Умножение ST(0) на целое число. ST(0)<-ST(0)*src. Операнд может быть 16- и 32-битным числом
FDIV FDIV ST(i) FDIV ST(i), SY	ST(0)<-ST(0)/ST(1) ST(0)<-ST(0)/ST(i) ST(i)<-ST(0)/ST(i)

Таблица П2.20 (окончание)

Команда	Описание
FDIVP ST(i), ST	Деление с выталкиванием из стека. ST(i)<-ST(0)/ST(i)
FIDIV src	Деление целых чисел. ST(0)<-ST(i)/src. Делитель может быть 16- и 32-битным числом
FDIVR ST(i), ST	Обратное деление вещественных чисел. ST(0)<-ST(i)/ST(0)
FDIVRP ST(i), ST	Обратное деление вещественных чисел и выталкивание из стека. ST(0)<-ST(i)/ST(0)
FIDIVR src	Обратное деление целых чисел. ST(0)<-src/ST(0)
FSQRT	Извлечь корень из ST(0) и поместить обратно
FSCALE	Масштабирование. ST(0)<-ST(0)*2^ST(1)
FXTRACT	Выделение мантиссы и порядка из числа ST(0). В ST(0) помещается порядок, в ST(1) — мантисса
FPREM	Нахождение остатка от деления. ST(0)<-ST(0)MODST(1)
FPREM1	Нахождение остатка от деления в стандарте IEEE
FRNDINT	Округление до ближайшего целого числа, находящегося в ST(0). ST(0)<-int(ST(0))
FABS	Нахождение абсолютного значения. ST(0)<-ABS(ST(0))
FCSH	Изменение знака ST(0)<- -ST(0)

Таблица П2.21. *Трансцендентные функции*

Команда	Описание
FCOS	Вычисление косинуса. ST(0)<-COS(ST(0)). Содержимое в ST(0) интерпретируется как угол в радианах
FPTAN	Частичный тангенс. Содержимое в ST(0) интерпретируется как угол в радианах. Значение тангенса возвращается на место аргумента, а затем в стек включается 1
FPATAN	Вычисление арктангенса. Вычисляется функция Arctg(ST(1)/ST(0)). После вычисления происходит выталкивание из стека, после чего результат оказывается в вершине стека
FSIN	Вычисление синуса. ST(0)<-SIN(ST(0)). Содержимое в ST(0) интерпретируется как угол в радианах
FSINCOS	Вычисление синуса и косинуса. ST(0)<-SIN(ST(0)) и ST(1)<-COS(ST(0))

Таблица П2.21 *(окончание)*

Команда	**Описание**
F2XM1	Вычисление 2^X-1. ST(0)<-2^ST(0)-1
FYL2X	Вычисление Y*LOG2(X). ST(0)=Y, ST(1)=X. Происходит выталкивание из стека, и только потом в вершину стека помещается результат вычисления
FYL2XP1	Вычисление Y*LOG2(X). ST(0)=Y, ST(1)=X. Происходит выталкивание из стека, и только потом в вершину стека помещается результат вычисления

Таблица П2.22. *Команды управления сопроцессором*

Команда	**Описание**
FINIT	Инициализация сопроцессора
FSTSW AX	Запись слова состояния в AX
FSTSW dest	Запись слова состояния в dest
FLDCW src	Загрузка управляющего слова (16 битов) из dest
FSTCW dest	Сохранение управляющего слова в dest
FCLEX	Сброс исключений
FSTENV dest	Сохранение состояния сопроцессора (SR, CR, TAGW, FIP, FDP) в памяти
FLDENV src	Загрузка состояния сопроцессора из памяти
FSAVE dest	Сохранение состояния сопроцессора и файла регистров в памяти
FRSTOR src	Загрузка состояния сопроцессора и файла регистров в памяти
FINCSTP	Инкремент указателя стека
FDECSTP	Декремент указателя стека
FFREE ST(i)	Освобождение регистра — пометка ST(i) как свободного
FNOP	Холостая операция сопроцессора
WAIT (FWAIT)	Ожидание процессором завершения текущей операции сопроцессора

Расширение MMX

Расширение MMX ориентировано в основном на использование в мультимедийных приложениях. Основная идея MMX заключается в одновременной обработке нескольких элементов данных за одну инструкцию. Расширение MMX появилось в процессорах модификации Pentium P54C и присутствует во всех последних модификациях этого процессора.

Расширение MMX использует новые типы упакованных данных: упакованные байты (восемь байтов), упакованные слова (четыре слова), упакованные двойные слова (два двойных слова), учетверенное слово. Расширение MMX включает восемь регистров общего пользования (MM0—MM7). Размер регистров составляет 64 бита. Физически эти регистры пользуются младшими битами рабочих регистров сопроцессора. Команды MMX "портят" регистр состояния и регистр тегов. По этой причине совместное использование команд MMX и команд сопроцессора может вызвать определенные трудности. Другими словами, перед каждым использованием команд MMX вам придется сохранять контекст сопроцессора, а это может весьма замедлить работу программы. Важно отметить также, что команды MMX работают непосредственно с регистрами сопроцессора, а не с указателями на элементы стека. В табл. П2.23 и П2.24 используются обозначения:

- mm — 64-битный регистр MMX;
- m32 и m64 — операнды, находящиеся в памяти и имеющие размер, соответственно, 32 и 64 бита;
- ir32 — обычные регистры процессора;
- imm — непосредственный операнд (константа) размером в 1 байт.

Таблица П2.23. *Команды MMX расширения (по книге [3])*

Команда	**Описание**
EMMS	Очистка стека регистров. Установка всех единиц в слове тегов
MOVD mm, m32/ir32	Пересылка данных в младшие 32 бита регистра MMX с заполнением старших битов нулями
MOVD m32/ir32, mm	Пересылка данных из младших 32 битов регистра MMX
MOVQ mm, mm/m64	Пересылка данных в регистр MMX
MOVQ mm/m64, mm	Пересылка данных из регистра MMX
PACKSSDW mm, mm/m64	Упаковка со знаковым насыщением двух двойных слов, расположенных в mm, и двух двойных слов mm/m64 в четыре слова, расположенных в mm

Таблица П2.23 *(продолжение)*

Команда	**Описание**
PACKSSWB mm, mm/m64	Упаковка со знаковым насыщением четырех слов, расположенных в mm, и четырех слов mm/m64 в восемь байтов, расположенных в mm
PACKUSWB mm, mm/m64	Упаковка с насыщением четырех знаковых слов, расположенных в mm, и четырех слов mm/m64 в восемь беззнаковых байтов, расположенных в mm
PADDB mm, mm/m64 PADDW mm, mm/m64 PADDD mm, mm/m64	Сложение упакованных байтов (слов или двойных слов) без насыщения (с циклическим переполнением)
PADDSB mm, mm/m64 PADDSW mm, mm/m64	Сложение упакованных байтов (слов) со знаковым насыщением
PADDUSB mm, mm/m64 PADDUSW mm, mm/m64	Сложение упакованных байтов (слов) с беззнаковым насыщением
PAND mm, mm/m64	Логическое "И"
PANDN mm, mm/m64	Логическое "И-НЕ"
PCMPEQB mm, mm/m64 PCMPEQD mm, mm/m64 PCMPEQW mm, mm/m64	Сравнение (на равенство) упакованных байтов (слов, двойных слов). Все биты элемента результата будут единичными (true) при совпадении соответствующих элементов операндов и нулевыми (false) — при несовпадении
PCMPGTB mm, mm/m64 PCMPGTD mm, mm/m64 PCMPGTW mm, mm/m64	Сравнение (по величине) упакованных знаковых байтов (слов, двойных слов). Все биты элемента результата будут единичными (true), если соответствующий элемент операнда назначения больше элемента операнда источника, и нулевыми (false) в противном случае
PMADDWD mm, mm/m64	Умножение четырех знаковых слов операнда-источника на четыре знаковых слова операнда-назначения. Два двойных слова результатов умножения младших слов суммируются и записываются в младшее двойное слово операнда-назначения. Два двойных слова результатов умножения старших слов суммируются и записываются в старшее двойное слово операнда-назначения
PMULHW mm, mm/m64	Умножение упакованных знаковых слов с сохранением только старших 16 битов элементов результата
PMULLW mm, mm/m64	Умножение упакованных знаковых или беззнаковых слов с сохранением только младших 16 битов результата

Таблица П2.23 *(продолжение)*

Команда	**Описание**
`POR mm, mm/m64`	Логическое "ИЛИ"
`PSHIMD mm, imm` `PSHIMQ mm, imm` `PSHIMW mm, imm`	`PSHIMD` представляет инструкции `PSLLD`, `PSRAD` и `PSRLD` с непосредственным операндом-счетчиком. `PSHIMW` представляет инструкции `PSLLW`, `PSRAW`, `PSRLW`. `PSHIMQ` представляет инструкции `PSLLQ` и `PSRLQ` с непосредственным операндом-счетчиком
`PSLLD mm, mm/m64` `PSLLQ mm, mm/m64` `PSLLW mm, mm/m64`	Логический сдвиг влево упакованных слов (двойных, учетверенных) операнда-назначения на количество битов, указанных в операнде-источнике, с заполнением младших битов нулями
`PSRAD mm, mm/m64` `PSRAW mm, mm/m64`	Арифметический сдвиг вправо упакованных двойных (учетверенных) знаковых слов операнда-назначения на количество битов, указанных в операнде-источнике, с заполнением младших битов битами знаковых разрядов
`PSRLD mm, mm/m64` `PSRLQ mm, mm/m64` `PSRLW mm, mm/m64`	Логический сдвиг вправо упакованных слов (двойных, учетверенных) операнда-назначения на количество битов, указанных в операнде-источнике, с заполнением старших битов нулями
`PSUBB mm, mm/m64` `PSUBW mm, mm/m64` `PSUBD mm, mm/m64`	Вычитание упакованных байтов (слов или двойных слов) без насыщения (с циклическим антипереполнением)
`PSUBSB mm, mm/m64` `PSUBSW mm, mm/m64`	Вычитание упакованных знаковых байтов (слов) с насыщением
`PSUBUSB mm, mm/m64` `PSUBUSW mm, mm/m64`	Вычитание упакованных беззнаковых байтов (слов) с насыщением
`PUNPCKHBW mm, mm/m64`	Чередование в регистре назначения байтов старшей половины операнда-источника с байтами старшей половины операнда-назначения
`PUNPCKHWD mm, mm/m64`	Чередование в регистре назначения слов старшей половины операнда-источника со словами старшей половины операнда-назначения
`PUNPCKHDQ mm, mm/m64`	Чередование в регистре назначения двойного слова старшей половины операнда-источника с двойным словом старшей половины операнда-назначения
`PUNPCKLBW mm, mm/m64`	Чередование в регистре назначения байтов младшей половины операнда-источника с байтами младшей половины операнда-назначения

Таблица П2.23 *(окончание)*

Команда	**Описание**
`PUNPCKLWD mm, mm/m64`	Чередование в регистре назначения слов младшей половины операнда-источника со словами младшей половины операнда-назначения
`PUNPCKLDQ mm, mm/m64`	Чередование в регистре назначения двойного слова младшей половины операнда-источника с двойным словом младшей половины операнда-назначения
`PXOR mm, mm/m64`	Исключающее "ИЛИ"

О новых инструкциях MMX

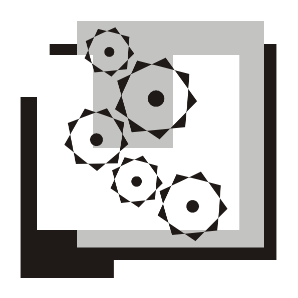
Перечисленные инструкции группы MMX с появлением Pentium 4 получили доступ к 128-битным регистрам (`xmm`). В табл. П2.24 перечислены новые MMX-инструкции.

Таблица П2.24. *Новые команды MMX*

Команда	**Описание**
`PADDQ xmm, xmm/m128`	Сложение двух 128-битных операндов
`PSUBQ xmm, xmm/m128`	Вычитание 128-битных операндов
`PMULUDQ xmm, xmm/m128`	Умножение 64-битных операндов, результат не должен превышать 128-битный размер
`PSLLDQ xmm, imm`	Логический сдвиг содержимого влево на `imm*8` битов
`PSRLDQ xmm, imm`	Логический сдвиг содержимого вправо на `imm*8` битов
`PSHUFHW xmm, xmm/m128, imm`	Пересылка с перегруппировкой четырех 16-битных слов из младшей половины `dest` в младшую половину `src`. Перегруппировка задается содержимым константы `imm`
`PSHUFLW xmm, xmm/m128, imm`	Пересылка с перегруппировкой четырех 16-битных слов из старшей половины `dest` в старшую половину `src`. Перегруппировка задается содержимым константы `imm`
`PSHUFD xmm, xmm/m128, imm`	Пересылка с перегруппировкой четырех 32-битных слов из `dest` в `src`. Перегруппировка задается содержимым константы `imm`
`PUNPCKHQDQ xmm, xmm/m128`	В `dest` записывается содержимое старших половин `src` и `dest`

Таблица П2.24 (окончание)

Команда	Описание
`PUNPCKLQDQ xmm, xmm/m128`	В `dest` записывается содержимое младших половин `src` и `dest`
`MOVDQ2Q mm, xmm`	Младшая половина `xmm` копируется в `mm`
`MOV2QDQ xmm, mm`	Содержимое регистра `mm` копируется в младшую половину `xmm`
`MOVNTDQ m128, xmm`	Пересылка содержимого 128-битного регистра в память без кэширования. Адрес должен быть кратен 16
`MOVDQA xmm, xmm/m128` `MOVDQA xmm/m128, xmm`	Команды пересылки 128-битного кода. Данные в памяти должны иметь адрес, кратный 16
`MOVDQU xmm, xmm/m128` `MOVDQU xmm/m128, xmm`	Команды пересылки 128-битного кода. Данные в памяти могут не иметь 16-битного выравнивания
`MOVMSKPD r32, xmm`	Копирует содержимое знаковых разрядов (63 и 127) в биты 0 и 1 регистра `r32`. Остальные биты регистра очищаются
`MASKMOVDQU xmm, xmm`	Пересылка по маске. Первый операнд содержит пересылаемый код, а второй операнд — маску пересылки. Адрес третьего (куда будет производиться пересылка) операнда должен находиться в `DS:DI` или в `DS:EDI`. Для каждого из 16 байтов выполняется следующее: если знаковый разряд `i`-го байта маски установлен, то `dest[i]=src[i]`; если знаковый разряд байта маски очищен, то содержимое `dest` не изменяется



Приложение 3

Защищенный режим микропроцессора Pentium

В *главах 3.6* и *4.6* говорится о схеме преобразования логического адреса в физический адрес и о функционировании в защищенном режиме вообще. В данном приложении представлена информация о некоторых структурах, используемых в этом режиме, и о некоторых общих положениях функционирования защищенного режима микропроцессора Intel.

Логический адрес в защищенном режиме складывается из смещения и селектора, который хранится в сегментном регистре. Селектор указывает (индексирует) на дескриптор, хранящийся в таблице дескрипторов. Дескриптор — это структура (см. далее), которая содержит линейный адрес начала сегмента. Вместе со смещением это дает линейный адрес конкретной ячейки памяти. Если в микропроцессоре включена еще и страничная адресация, то получившийся линейный адрес подвергается еще дополнительному преобразованию *(см. главу 3.6)*. О структурах страничной адресации поговорим далее.

Об уровнях привилегий

Уровни привилегий нумеруются от 0 до 3. Номер 3 является самым низким уровнем привилегий. Нулевой привилегией обладает ядро операционной системы. Уровни привилегий относятся к дескрипторам, селекторам и задачам. В регистре флагов имеется поле привилегий ввода/вывода, которое регулирует управление доступом к инструкциям ввода/вывода (`IOPL`). Уровень привилегий задачи определяется двумя младшими битами сегмента `CS`.

При страничной адресации имеются всего два уровня доступа — 3 и 0.

Селекторы

В отличие от реального режима, сегментные регистры содержат в защищенном режиме не адреса, а селекторы. Рассмотрим структуру селектора:

- биты 0 и 1 — запрошенный программой уровень привилегий;
- 2-й бит определяет, использовать глобальную таблицу дескрипторов GDT (0) или локальную таблицу дескрипторов LDT (1);
- биты 3—16 — индекс дескриптора в таблице.

Дескриптор для защищенного режима — это 64-битная структура, которая может описывать сегмент кода, сегмент данных, сегмент состояния задачи, шлюз вызова, ловушки, прерывания или задачи. Дескриптор в глобальной дескрипторной таблице может описывать локальную дескрипторную таблицу.

Дескриптор кода и данных

На рис. П3.1 изображена структура дескриптора кода и данных.

Биты 24–31 базы сегмента	Биты доступа	Базовый адрес, 24 бита	Предел, 16 битов

Рис. П3.1. Структура дескриптора кода и данных

Базовый адрес сегмента содержит физический адрес сегмента. Предел содержит размер сегмента в байтах, уменьшенный на единицу.

Описание других битов дескриптора:

- 6-й байт:
  - биты 0—3 определяют биты 16—19 предела;
  - бит 4 зарезервирован для операционной системы;
  - бит 5 равен 0;
  - бит 6 — разрядность (0 — 16-битный, 1 — 32-битный);
  - бит 7 — гранулярность (0 — лимит в байтах, 1 — лимит в 4-килобайтных величинах).
- 5-й байт:
  - бит 0 — если 1, то к сегменту было обращение;
  - бит 1 — разрешение чтения для кода, записи для данных;
  - бит 2 — бит подчиненности для кода, бит расширения для данных;

- бит 3 — тип сегмента (0 — данные, 1 — код);
- бит 4 — тип дескриптора (1 — не системный);
- биты 5 и 6 — уровень привилегий дескриптора;
- бит 7 — бит присутствия сегмента.

Другие дескрипторы

Если в дескрипторе бит 4 (в 5-м байте) равен 0, то дескриптор называется *системным*. В этом случае биты 0—3 определяют один из возможных типов дескрипторов:

- 0 — зарезервированный тип;
- 1 — свободный 16-битный TSS (TSS — сегмент состояния задачи);
- 2 — дескриптор таблицы LDT. Данный дескриптор хранится в GDT, т. е. глобальной дескрипторной таблице;
- 3 — занятый 16-битный TSS;
- 4 — 16-битный шлюз вызова;
- 5 — шлюз задачи;
- 6 — 16-битный шлюз прерывания;
- 7 — 16-битный шлюз ловушки;
- 8 — зарезервировано;
- 9 — свободный 32-битный TSS;
- 10 — зарезервировано;
- 11 — занятый 32-битный TSS;
- 12 — 32-битный шлюз вызова;
- 13 — зарезервировано;
- 14 — 32-битный шлюз прерывания;
- 15 — 32-битный шлюз ловушки.

Команды CALL или JMP на адрес с селектором, указывающим на дескриптор шлюза, осуществляют передачу управления по адресу, указанному в дескрипторе. Если селектор указывает на шлюз задачи, то это приводит к переключению задач. Обычные же переходы JMP, CALL, RET, IRET возможны лишь к сегментам с тем же уровнем привилегий либо более низким уровнем привилегий.

Сегмент TSS

Сегмент TSS (Task State Segment) — сегмент задачи, используется для хранения контекста задачи. Селектор данного сегмента хранится в регистре `TR`. Дескриптор же данного сегмента, на который указывает селектор, хранится в глобальной дескрипторной таблице (GDT). Предполагается, что операционная система должна сохранять все данные о задаче в этом сегменте перед тем, как переключиться на другую задачу. В частности, там хранятся значения всех регистров задачи и битовая карта, которая определяет, какие команды ввода/вывода можно выполнять данной задаче, вопреки значению поля `IOPL` в регистре флагов.

О защите и уровнях привилегий

В защищенном режиме на уровне сегментов принята трехуровневая схема защиты. Самые большие привилегии соответствуют уровню 0, минимальные — 3. Код программы, хранящийся в некотором сегменте, имеет уровень привилегий этого сегмента. Соответственно данная программа может обратиться только к сегменту, имеющему такой же уровень привилегий или меньший уровень. Это относится как к сегменту данных, так и к сегменту, где хранится код. Последнее предполагает попытку вызвать из сегмента какую-либо процедуру или переход на метку в этом сегменте.

Между уровнем привилегий, определенных в селекторе (RPL, Requested Privilege Level), и уровнем привилегий, определенных в дескрипторе (DPL, Descriptor Privilege Level), существует следующая взаимосвязь. Уровень привилегий, определенный в селекторе, может только уменьшить уровень привилегий задачи, который определен в дескрипторе сегмента, где задача расположена. Другими словами, если уровень привилегий в селекторе равен 0, то реальный уровень привилегий задачи определяется из уровня привилегий дескриптора сегмента задачи. Если же вы задаете в селекторе уровень привилегий равным 3, то реальный уровень привилегий (CPL, Current Privilege Level) будет определяться значением RPL.

Привилегированные команды

Кроме команд, которые могут выполняться программами с любой степенью привилегий и которых большинство, имеются команды, выполнять которые могут только программы с достаточным уровнем привилегий.

К первой группе относятся команды, воздействующие на механизмы сегментации и защиты. К таким командам относятся `HLT`, `CLTS`, `LGDT`, `LIDT`, `LLDT`, `LTR`,

LMSW и др. В эту же группу входят команды передачи данных, в которых получателем или источником являются регистры управления CR*n*, регистры отладки DR*n*, регистры проверки TR*n*.

Ко второй группе относятся команды ввода/вывода IN, OUT, INS, OUTS. А также команды CLI и STI, действующие на флаг прерывания. Программа, не имеющая привилегии 0, может выполнять эти команды ввода/вывода, если установлены соответствующие биты в сегменте задачи TSS. Биты определяют адреса, по которым можно осуществлять операции ввода/вывода. Кроме этого, доступ к командам второй группы определяется соотношением флага IOPL и текущего уровня привилегированности задачи (CPL). Если текущий уровень привилегированности меньше или равен IOPL, то задача может выполнять любые команды ввода/вывода.

Переключение задач

Состояние каждой задачи (значение всех регистров, связанных с данной задачей) хранится в сегменте состояния задачи, на который указывает адрес в регистре задачи TR. При переключении задач достаточно загрузить новый селектор в регистр задачи, и состояние старой задачи автоматически сохранится в TSS, в процессор же загрузится состояние новой задачи.

Страничное управление памятью

Механизм страничного управления памятью включается установкой бита PG в регистре CR0. Регистр CR2 хранит линейный адрес отказа и адрес памяти, по которому был обнаружен последний отказ страницы. Регистр CR3 хранит физический адрес каталога страниц. Младшие 12 битов этого регистра всегда равны нулю (выравнивание по границе страниц). Каталог страниц состоит из 32-битных элементов и имеет длину 4 Кбайт. Структура элемента каталога представлена на рис. П3.2.

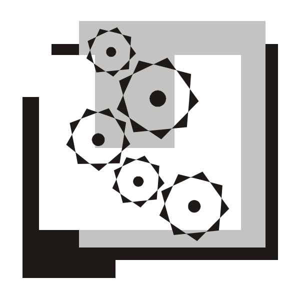
20 старших битов адреса таблицы следующего уровня	Резерв 3 бита	G	PS	D	A	PCD	PWT	U/S	R/W	P

Рис. П3.2. Структура элемента каталога

Каждая таблица страниц также имеет размер 4 Кбайт и элементы аналогичного формата. Но эти элементы содержат базовый адрес самих страниц и ат-

рибуты страниц. Физический адрес собирается из базового адреса и младших 12 битов линейного адреса. Значение атрибутов страниц:

- G — глобальная страница, страница не удаляется из буфера;
- PS — размер страницы; если 1, то размер страницы равен 2 или 4 Мбайт, если 0, то размер другой;
- D — грязная (занятая) страница. Устанавливается в 1 при записи на страницу;
- A — бит доступа. Устанавливается в 1 при любом обращении к странице;
- PCD — бит запрещения кэширования;
- PWT — бит разрешения сквозной записи;
- U/S — страница или таблица доступна для программ с уровнем доступа 3;
- R/W — страница/таблица доступна для записи;
- P — страница/таблица присутствуют.



Приложение 4

Структура исполняемых модулей

Исполняемым форматом в Windows является формат PE. Сокращение PE означает Portable Executable, т. е. переносимый исполняемый формат. Этот формат имеют как exe-файлы, так и динамические библиотеки. Важно, что сейчас фирма Microsoft ввела "новый" формат и для объектных модулей — это COFF-формат (Common Object File Format), который, однако, на поверку оказался, в сущности, все тем же PE-форматом. Замечу в этой связи, что фирма Borland по-прежнему работает с объектными файлами, имеющими структуру OMF (Object Module Format). Старый NE-формат (New Executable), используемый старой операционной системой Windows и рассчитанный на сегментную структуру памяти, ушел в небытие. Кроме того, есть еще формат V*x*D-драйверов — LE-формат (Linear Executable, линейный исполняемый). Таким образом, данное приложение будет посвящено разбору структуры исполняемых PE-модулей.

Поскольку в состав исполняемого PE-модуля входит и DOS-программа (stub), мы начнем наше рассмотрение со структуры DOS-программ. Наше рассмотрение будет кратким, и мы воспользуемся таблицей из [1] — табл. П4.1.

Таблица П4.1. *Структура exe-программы для MS-DOS*

Смещение	Длина	Название	Описание
+0	2	"MZ"	Подпись, признак exe-программы
+2	2	`PartPag`	Длина неполной последней страницы
+4	2	`PageCnt`	Длина в страницах (по 512 байтов), включая заголовок и последнюю страницу
+6	2	`ReloCnt`	Число элементов в таблице перемещения
+8	2	`HdrSize`	Длина заголовка в параграфах

Таблица П4.1 (окончание)

Смещение	Длина	Название	Описание
+0AH	2	`MinMem`	Минимум требуемой памяти за концом программы
+0CH	2	`MaxMem`	Максимум требуемой памяти за концом программы
+0EH	2	`ReloSS`	Сегментный адрес стека
+10H	2	`EXESp`	Значение регистра `SP`
+12H	2	`ChkSum`	Контрольная сумма
+14H	2	`ExeIP`	Значение регистра `IP`
+16H	2	`ReloCS`	Сегментный адрес кодового сегмента
+18H	2	`TablOff`	Смещение в файле первого элемента таблицы перемещения
+1AH	2	`Overlay`	Номер оверлея, 0 для главного модуля
* Конец форматированной порции заголовка **			
+1CH			
** Начало таблицы перемещения (возможно с 1CH) **			
+?		4*?	смещение сегмент...смещение сегмент

Более подробный разбор структуры заголовка DOS-программы можно найти в [1]. Добавлю только, что сразу за таблицей перемещения начинается исполняемая часть модуля. Таблица перемещения используется для того, чтобы при загрузке настроить ссылки на адреса сегментов. Это необходимо лишь в том случае, если в программе используются адреса сегментов. В противном случае таблица перемещения не содержит элементов, так что исполняемый код начинается сразу за форматированной частью заголовка. Перейдем теперь к общей структуре PE-модуля.

Общая структура PE-модуля

Начало заголовка exe-файлов в Win32 представляет собой небольшую DOS-программу[1], основное предназначение которой заключается в том, чтобы при запуске в операционной системе MS-DOS сделать сообщение о том, что дан-

[1] Подробнее о заголовке PE-модуля см. [27].

ный модуль не предназначен для работы в MS-DOS. Программа LINK.EXE (TLINK32.EXE) устанавливает свой вариант DOS-программы. Однако при желании вы всегда можете поставить свою программу-заглушку (от англ. *stub* — заглушка).

Рассмотрим общую структуру PE-модуля (табл. П4.2).

Таблица П4.2. *Общая структура PE-модуля*

Смещение	Описание
`00H`	Стандартный DOS-заголовок
`1CH`	Четыре байта для выравнивания до 20H байтов (до границы двойного параграфа)
`20H`	Информация о программе, обычно отсутствующая
`3CH`	Смещение 32-битного PE-заголовка
`40H`	Таблица перемещения для программы-заглушки. У стандартных заглушек эта таблица, разумеется, пуста. Тем не менее, указатель `TablOff` должен показывать именно сюда
`40H`+??	Здесь начинается тело самой заглушки, которая следует за таблицей перемещения. Естественно, в стандартных заглушках нет ничего, кроме сообщения о невозможности запуска программы в операционной системе MS-DOS
??	Здесь начинается собственно PE-заголовок. Сюда показывает содержимое четырех байтов по адресу `3CH`. Начало должно быть выровнено по 8-байтной границе
??	Таблица описаний секций файлов (Object Table)
??	Остальная информация: COFF-символы, отладочная информация, таблица импорта и таблица экспорта, ресурсы и т. д. Данный раздел называется Image Pages, т. е. страницы образов

В листинге П4.1 показан фрагмент PE-заголовка. Обратите внимание, что по смещению `3CH` действительно находится адрес начала основного заголовка (символы `PE`).

Листинг П4.1. Фрагмент PE-заголовка

```
00000: 4D 5A 0A 00 02 00 00 00 ¦ 04 00 0F 00 FF FF 00 00 MZ
00010: C0 00 00 00 00 00 00 00 ¦ 40 00 00 00 00 00 00 00
00020: 00 00 00 00 00 00 00 00 ¦ 00 00 00 00 00 00 00 00
00030: 00 00 00 00 00 00 00 00 ¦ 00 00 00 00 80 00 00 00
00040: B4 09 BA 10 00 0E 1F CD ¦ 21 B8 01 4C CD 21 90 90
00050: 54 68 69 73 20 69 73 20 ¦ 61 20 57 69 6E 33 32 20
```

```
00060: 70 72 6F 67 72 61 6D 2E ¦ 0D 0A 24 00 00 00 00 00
00070: 00 00 00 00 00 00 00 00 ¦ 00 00 00 00 00 00 00 00
00080: 50 45 00 00 4C 01 03 00 ¦ 39 30 00 00 00 00 00 00 PE
00090: 00 00 00 00 E0 00 0E 03 ¦ 0B 01 02 34 00 30 00 00
000A0: 00 10 00 00 00 60 00 00 ¦ 30 96 00 00 00 70 00 00
000B0: 00 A0 00 00 00 00 40 00 ¦ 00 10 00 00 00 02 00 00
```

Заголовок PE-модуля

В табл. П4.3 приведено описание заголовка PE-модуля.

Таблица П4.3. *Заголовок PE-модуля*

Смещение	Длина поля	Название поля	Описание поля
00h	DWORD	Signature Bytes	Сигнатура. Первые два байта "PE" 4550H. Еще два байта обязательно должны быть равны нулю
04h	WORD	CPU Type	Данное поле указывает на процессор, который следует предпочесть при запуске программы. Вот возможные значения этого поля: • 0000h — неизвестный процессор; • 014Ch — i386; • 014Dh — i486; • 014Eh — i586; • 0162h — MIPS Mark I (R2000, R3000); • 0163h — MIPS Mark II (R6000); • 0166h — MIPS Mark III (R4000). Чаще всего данное поле указывает на процессор 386
06h	WORD	Num of Objects	Поле указывает на число реальных входов в Object Table (см. табл. П4.4)
08h	DWORD	Time/Date Stamp	Дата и время, которые устанавливаются при компоновке программы
0Ch	DWORD	Pointer to COFF table	Дополнительный указатель, определяющий местонахождение отладочной COFF-таблицы. Это поле используется только в OBJ-файлах и PE-файлах, содержащих отладочную COFF-информацию
10h	DWORD	COFF table size	Количество символов в COFF-таблице

Таблица П4.3 *(продолжение)*

Смеще-ние	Длина поля	Название поля	Описание поля
14h	WORD	NT Header Size	Размер заголовка PE-файла, начиная с поля Magic — таким образом, общий размер заголовка PE-файла составляет NT Header Size + 18h
16h	WORD	Flags	Указывает на предназначение программы. Значение флагов: • 0000h — это программа; • 0001h — файл не содержит таблицы перемещений; • 0002h — образ в файле можно запускать на выполнение. Если этот бит не установлен, то это обычно указывает на ошибку, обнаруженную на этапе компоновки, или же на то, что код был скомпонован инкрементально (инкрементальная компоновка — это частичная компоновка кода при изменении участка программы, вместо полной перекомпиляции проекта); • 0200h — загружать в память по фиксированному адресу. Указывает на то, что программу можно загрузить только по адресу, записанному в Image Base, если это невозможно, то такой файл лучше вообще не запускать; • 2000h — это библиотека
18h	WORD	Magic	Слово сигнатуры, определяющее состояние отображенного файла. Возможные значения: • 107h — отображение ПЗУ; • 10Bh — нормально исполняемое отображение
1Ah	BYTE	Link Major	Старший номер версии использовавшегося при создании модуля компоновщика, в двоично-десятичном коде
1Bh	BYTE	Link Minor	Младший номер версии использовавшегося при создании модуля компоновщика, в двоично-десятичном коде

Таблица П4.3 *(продолжение)*

Смещение	**Длина поля**	**Название поля**	**Описание поля**
`1Ch`	`DWORD`	Size of Code	Размер собственно программного кода в файле. KERNEL использует это значение для распределения памяти под загружаемую программу. Установка этого значения слишком маленьким приведет к выдаче сообщения о нехватке памяти. Обычно большинство модулей имеют только одну программную секцию — `.text`
`20h`	`DWORD`	Size of Init Data	Размер секции инициализированных данных, очевидно, не используется в Windows 9x, но используется в Windows 2000 и выше. Назначение аналогично приведенному ранее
`24h`	`DWORD`	Size of UnInit Data	Размер секции неинициализированных данных. Неинициализированные данные обычно содержатся в секции `.bss`. Эта секция не занимает на диске никакого места, но при загрузке модуля загрузчик отводит под нее память
`28h`	`DWORD`	Entry point RVA	Адрес относительно Image Base, по которому передается управление при запуске программы или адрес инициализации/завершения библиотеки
`2Ch`	`DWORD`	Base of Code	Адрес секции относительно базового адреса (`40000H`), содержащей программный код. Этот адрес обычно равен `1000H` для компоновщика Microsoft и `10000H` для компоновщика Borland
`30h`	`DWORD`	Base of Data	Адрес относительно базового (`40000H`), с которого начинаются секции данных файла. Секции данных обычно идут последними в памяти, после заголовка PE и программных секций
`34h`	`DWORD`	Image Base	При создании компоновщик помещает сюда адрес, по которому будет отображен исполняемый файл в памяти. Если загрузчик отобразит файл именно по этому адресу, то дополнительной настройки не потребуется

Таблица П4.3 *(продолжение)*

Смещение	**Длина поля**	**Название поля**	**Описание поля**
38h	DWORD	Object align	Выравнивание программных секций. После отображения в память каждая секция будет обязательно начинаться с виртуального адреса, кратного данной величине
3Ch	DWORD	File align	В случае PE-файла исходные данные, которые входят в состав каждой секции, будут обязательно начинаться с адреса, кратного данной величине. Значение по умолчанию составляет 200H
40h	WORD	OS Major	Старший номер версии операционной системы, необходимой для запуска программы
42h	WORD	OS Minor	Младший номер версии операционной системы
44h	WORD	USER Major	Пользовательский номер версии, задается пользователем при компоновке программы. Старшая часть
46h	WORD	USER Minor	Пользовательский номер версии, младшая часть
48h	WORD	SubSys Major	Старший номер версии подсистемы
4Ah	WORD	SubSys Minor	Младший номер версии подсистемы. Типичное значение версии 4.0, что означает Windows 95
4Ch	DWORD	Reserved	Зарезервировано
50h	DWORD	Image Size	Представляет общий размер всех частей отображения, находящихся под контролем загрузчика. Эта величина равна размеру области памяти, начиная с базового адреса отображения и заканчивая адресом конца последней секции. Адрес конца секции выровнен на ближайшую верхнюю границу секции
54h	DWORD	Header Size	Общий размер всех заголовков: DOS Stub + PE Header + Object Table

Таблица П4.3 (продолжение)

Смещение	Длина поля	Название поля	Описание поля
58h	DWORD	File CheckSum	Контрольная сумма всего файла. Как и в операционной системе MS-DOS, ее никто не контролирует, а компоновщик устанавливает ее в 0. Предполагалось ее рассчитывать как инверсию суммы всех байтов файла
5Ch	WORD	SubSystem	Операционная подсистема, необходимая для запуска данного файла. Вот значения этого поля: • 1 — подсистема не требуется (NATIVE); • 2 — запускается в подсистеме Windows GUI; • 3 — запускается в подсистеме Windows character (терминальное или консольное приложение); • 5 — запускается в подсистеме OS/2; • 7 — запускается в подсистеме POSIX
5Eh	WORD	DLL Flags	Определяет дополнительные требования при загрузке, начиная с операционной системы Windows NT 3.5. Устарел и не используется
60h	DWORD	Stack Reserve Size	Память, требуемая для стека приложения. Память резервируется, но выделяется только Stack Commit Size байтов. Следующая страница является охранной. Когда приложение достигает этой страницы, то она становится доступной, а следующая страница — охранной, и так до достижения нижней границы, после чего Windows удаляет программу
64h	DWORD	Stack Commit Size	Объем памяти, отводимой для стека сразу после загрузки
68h	DWORD	Heap Reserve Size	Максимально возможный размер локальной кучи
6Ch	DWORD	Heap Comit Size	Размер кучи, распределяемый при загрузке

Таблица П4.3 *(продолжение)*

Смещение	Длина поля	Название поля	Описание поля
70h	DWORD	Loader Flags	Начиная с Windows NT 3.5, объявлено неиспользуемым, назначение неясно, но в целом связано с поддержкой отладки
74h	DWORD	Num of RVA and Sizes	Указывает размер массива VA/Size, который следует ниже, данное поле зарезервировано под будущие расширения формата. В данный момент его значение всегда равно 10h
78h	DWORD	Export Table RVA	Относительный адрес (относительно базового адреса) таблицы экспорта
7Ch	DWORD	Export Data Size	Размер таблицы экспорта
80h	DWORD	Import Table RVA	Относительный адрес (относительно базового адреса) таблицы импорта
84h	DWORD	Import Data Size	Размер таблицы импорта
88h	DWORD	Resource Table RVA	Относительный адрес (относительно базового адреса) таблицы ресурсов
8Ch	DWORD	Resource Data Size	Размер таблицы ресурсов
90h	DWORD	Exception Table RVA	Относительный адрес таблицы исключений
94h	DWORD	Exception Data Size	Размер таблицы исключений
98h	DWORD	Security Table RVA	Адрес таблицы безопасности. По-видимому, не используется
9Ch	DWORD	Security Data Size	Размер таблицы безопасности
A0h	DWORD	Fix Up's Table RVA	Относительный адрес таблицы настроек
A4h	DWORD	Fix Up's Data Size	Размер таблицы настроек
A8h	DWORD	Debug Table RVA	Относительный адрес таблицы отладочной информации

Таблица П4.3 (окончание)

Смещение	Длина поля	Название поля	Описание поля
ACh	DWORD	Debug Data Size	Размер таблицы отладочной информации
B0h	DWORD	Image Description RVA	Относительный адрес строки описания модуля
B4h	DWORD	Description Data Size	Размер строки описания модуля
B8h	DWORD	Machine Specific RVA	Адрес таблицы значений, специфичных для микропроцессора
BCh	DWORD	Machnine Data Size	Размер таблицы значений, специфичных для микропроцессора
C0h	DWORD	TLS RVA	Указатель на локальную область данных потоков
C4h	DWORD	TLS Data Size	Размер области данных потоков
C8h	DWORD	Load Config RVA	Назначение неизвестно
CCh	DWORD	Load Config Data Size	Назначение неизвестно
D0h	08h	Reserved	Зарезервировано
D8h	DWORD	IAT RVA	Используется в Windows 2000 и далее, в Windows 9*x*, судя по всему, нет
DCh	DWORD	IAT Data Size	Размер описанного поля
E0h	08h	Reserved	Зарезервировано
E8h	08h	Reserved	Зарезервировано
F0h	08h	Reserved	Зарезервировано

Таблица секций

Между заголовком PE-модуля и данными для секций расположена таблица секций, элемент которой представлен в табл. П4.4. Элемент таблицы секций содержит полную информацию об одной секции.

Таблица П4.4. *Элемент таблицы секций*

Смещение	Длина поля	Название поля	Описание поля
`00h`	`08h`	Object Name	Имя объекта, остаток заполнен нулями. Если имя объекта имеет длину 8 символов, то заключительного 0 нет. Вот несколько возможных имен: • `.text` — исполняемый код общего назначения; • `CODE` — исполняемый код, формируемый компоновщиками фирмы Borland; • `.icode` — переходники (инструкции `JMP`), помещаемые сюда старой версией TLINK32; • `.data` — инициализированные данные, помещаются компоновщиком фирмы Microsoft; • `DATA` — инициализированные данные, помещаемые сюда компоновщиком TLINK32; • `.bss` — неинициализированные глобальные и статические переменные; • `.CRT` — еще одна секция для хранения инициализированных данных; • `.rsrc` — секция для хранения ресурсов; • `.idata` — секция импорта; • `.edata` — секция экспорта; • `.reloc` — секция настроек. Данная информация может понадобиться загрузчику, если он не сможет загрузить модуль по базовому адресу; • `.tls` — данные для запуска потоков; • `.rdata` — данная секция в основном содержит отладочную информацию; • `.debug$s` и `.debug$t` — данные секции есть только в COFF-объектных файлах. Они содержат информацию о символах CodeView и их типах; • `.drective` — в данной секции содержится текст программ для компоновки. Эта секция есть только в объектных файлах. Секции, содержащие символ `$`, обрабатываются особым образом. Компоновщик объединяет все секции, имеющие одинаковые символы в имени до символа `$`. Это имя (до символа `$`) присваивается полученной секции

Таблица П4.4 *(продолжение)*

Смещение	Длина поля	Название поля	Описание поля
08h	DWORD	Virtual Size	Виртуальный размер секции — именно столько памяти будет отведено под секцию. Если Virtual Size превышает Physical Size, то разница заполняется нулями: так определяются секции неинициализированных данных (Physical Size = 0)
0Ch	DWORD	Section RVA	Размещение секции в памяти, ее виртуальный адрес относительно Image Base. Адрес каждой секции должен быть кратен значению, хранящемуся в Object Align (диапазон: степень 2 от 512 до 256 Мбайт включительно, по умолчанию 64 Кбайта), секции обычно упакованы вплотную друг к другу, что, впрочем, не является обязательным. Для объектных файлов поле не имеет смысла
10h	DWORD	Physical Size	Размер секции (ее инициализированной части) в файле. Значение поля Physical Size должно быть кратно значению поля File align в заголовке PE Header и, кроме того, должно быть меньше или равно Virtual Size. Для объектных файлов поле содержит точный размер секции, сгенерированный компилятором или ассемблером. Другими словами, для объектных файлов Physical Size эквивалентно Virtual Size
14h	DWORD	Physical Offset	Физическое смещение относительно начала exe-файла, выровнено относительно фактора File align, указанного в заголовке модуля
18h	DWORD	Pointer to Linenumber	Файловое смещение таблицы номеров строк. Используется для объектных файлов
1Ch	WORD	Number of Relocations	Количество перемещений в таблице поправок. Используется только для объектных файлов
1Eh	WORD	Number of Linenumbers	Количество номеров строк в таблице номеров строк для данной секции. Используется для объектных файлов
20h	08h	Reserved	Зарезервировано для объектных файлов

Таблица П4.4 *(окончание)*

Смещение	Длина поля	Название поля	Описание поля
`28h`	`DWORD`	Object Flags	Битовые флаги секции: • `00000004h` — используется для кода с 16-битными смещениями; • `00000020h` — секция кода; • `00000040h` — секция инициализированных данных; • `00000080h` — секция неинициализированных данных; • `00000200h` — комментарии или любой другой тип информации; • `00000400h` — оверлейная секция; • `00000800h` — не будет являться частью образа программы; • `00001000h` — общие данные; • `00500000h` — выравнивание по умолчанию, если не указано иное; • `02000000h` — может быть выгружен из памяти; • `04000000h` — не кэшируется; • `08000000h` — не подвергается страничному преобразованию; • `10000000h` — разделяемый; • `20000000h` — выполнимый; • `40000000h` — можно читать; • `80000000h` — можно писать

Рассмотрим некоторые наиболее важные секции PE-модуля.

Секция экспорта (*.edata*)

Секция экспорта (`.edata`) состоит из следующих таблиц:

- таблица собственно экспорта (Export Directory Table);
- адресная таблица (Export Address Table);
- таблица указателей на имена (Export Name Table Pointers);
- таблица номеров (Export Ordinal Table);
- таблица самих имен (Export Name Table).

Структура таблицы экспорта показана в табл. П4.5.

Таблица П4.5. *Таблица экспорта*

Смещение	**Длина поля**	**Название поля**	**Описание поля**
00h	DWORD	Flags	Зарезервировано, должно быть равно нулю
04h	DWORD	Time/Date Stamp	Время и дата создания экспортных данных
08h	WORD	Major Version	Старший номер версии таблицы экспорта. Не используется
0Ah	DWORD	Minor Version	Младший номер версии таблицы экспорта, также не используется
0Ch	DWORD	Name RVA	Относительный адрес строки, в которой указано имя модуля библиотеки
10h	DWORD	Ordinal Base	Начальный номер экспорта, для функций, экспортируемых данным модулем
14h	DWORD	Num of Functions	Количество функций, экспортируемых данным модулем, является числом элементов массива Address Table *(см. далее)*
18h	DWORD	Num of Name Pointers	Число указателей на имена, обычно равно числу функций (но это не так, если у нас есть функции, экспортируемые только по номеру)
1Ch	DWORD	Address Table RVA	Указатель на таблицу относительных адресов экспортируемых функций
20h	DWORD	Name Pointers RVA	Указатель на таблицу указателей на имена экспортируемых функций данного модуля
24h	DWORD	Ordinal Table RVA	Указатель на таблицу номеров экспорта, данный массив по индексам параллелен Name Pointers, элементами являются слова

Таблица адресов экспорта. Эта структура данных содержит адреса экспортируемых функций (их точки входа) в формате DWORD (по 4 байта на элемент). Для доступа к данным используется номер функции с коррекцией на базу номеров (Ordinal Base).

Таблица указателей на имена. Данная структура содержит указатели на имена экспортируемых функций, указатели отсортированы в лексическом порядке

для обеспечения возможности бинарного поиска. Каждый указатель занимает 4 байта. Имена функций обычно лежат в секции экспорта.

Таблица номеров. Данная структура совместно с Name Table Pointers формирует два параллельных массива, разделенных для облегчения к ним доступа индексированием на родные для процессора данные (слова, двойные слова, но не сложные структуры). Данный массив содержит номера экспорта, которые в общем случае являются индексами в Address Table экспорта (за вычетом базы Ordinal Base). Элементами данного массива являются слова (2 байта).

Таблица имен экспорта. Эта таблица содержит необязательные (по мнению Microsoft) имена экспортируемых функций. Данный массив используется совместно с Name Table Pointers и Ordinal Table для обеспечения связывания загрузчиком импорта/экспорта по имени. Механизм описывался выше. Каждый элемент являет собой ASCIIZ-строку с именем экспортируемой функции. Никто не говорит, что они должны в файле идти друг за другом последовательно, хотя так построено большинство файлов. Надо отметить, что имена экспорта чувствительны к регистру. Отмечу особенность загрузчика — при связывании, если адрес функции находится в секции экспорта, на самом деле по указанному адресу лежит строка, переадресующая к другой библиотеке, экспортирующей данную функцию (с указанием библиотеки и самой функции). Это называется *передачей экспорта.*

Секция импорта (*.idata*)

Схема вызова импортируемых функций из PE-модуля изображена на рис. П4.1, которая с некоторыми изменениями взята из [2]. Смысл данного рисунка заключается в следующем. При компоновке все вызовы API-функций преобразуются к вызову типа `CALL` *`Адрес1`*. При этом адрес, так же как и вызов, находится в секции кода (`.text`). По адресу же стоит команда

```
JMP DWORD PTR [Адрес2]
```

`[`*`Адрес2`*`]` находится в секции `.idata` (импорта) и содержит двойное слово — адрес функции в динамической библиотеке. Современные компиляторы содержат директивы, позволяющие вместо двух вызовов (`CALL` и `JMP`) генерировать один — `CALL [`*`Адрес2`*`]`.

Секция импорта состоит из следующих таблиц:

- каталог импорта (Import Directory Table):
- таблица ссылок на имена (LookUp Table);
- таблица имен сервисов (Hint-Name Table);
- таблица адресов импорта (Import Address Table).

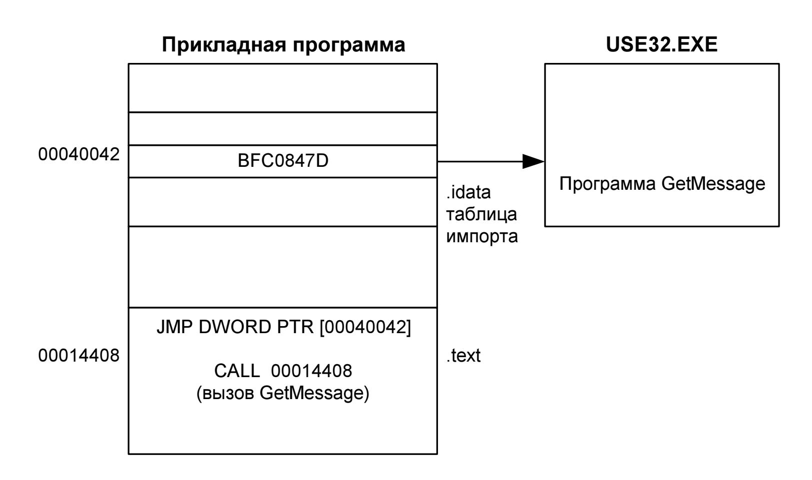


Рис. П4.1. Вызов импортируемой функции

Каталог импорта состоит из элементов, структура которых приведена в табл. П4.6.

Таблица П4.6. *Элемент каталога импорта*

Смещение	Длина поля	Название поля	Описание поля
00h	DWORD	Import LookUp	Содержит ссылку на таблицу относительных адресов (относительно базового адреса), указывающих на соответствующие имена импортируемой функции, или непосредственно номер импортируемого входа
04h	DWORD	Time/Date Stamp	Отметка о времени создания часто содержит ноль
08h	DWORD	Forward Chain	Связано с возможностью передачи экспорта в другие библиотеки. Обычно равно `0FFFFFFFFh`
0Ch	DWORD	Name RVA	Ссылка на библиотеку импорта в виде ASCII-строки с нулем на конце. Например, KERNEL32.DLL или USER32.DLL
10h	DWORD	Address Table RVA	Ссылка на таблицу адресов импорта, заполняется системой при связывании

В табл. П4.7 приведена структура таблицы просмотра импорта, или таблицы имен сервисов. В таблице имен сервисов имеется ссылка из поля Import LookUp на массив, содержащий ссылки на таблицу просмотра импорта. При импортировании по номеру старший бит элемента массива равен 1.

Таблица П4.7. *Таблица имен сервисов*

Тип	Содержимое
`Word`	Номер функции
`Hint`	ASCII-имя функции

Таблица адресов импорта принимает в себя информацию после связывания загрузчиком импорта из внешних библиотек, она завершается нулевым элементом.

Локальная область данных потоков

Локальная область данных потоков — это специальный, протяженный блок данных. Каждый поток при его создании получает собственный блок локальных данных.

Локальные области данных потоков представлены следующими таблицами:

- таблица разделов потоков (TLS Directory Table);
- данные потоков (TLS Data);
- индексные переменные (Index Variables);
- адреса обратных вызовов (callback).

Структура таблицы разделов потоков представлена в табл. П4.8.

Таблица П4.8. *Таблица разделов потоков*

Смещение	Длина поля	Название поля	Описание поля
`00h`	`DWORD`	Start Data Block VA	Виртуальный адрес начала блока данных потока
`04h`	`DWORD`	End Data Block VA	Виртуальный адрес конца блока данных потока
`08h`	`DWORD`	Index VA	Виртуальный адрес индексной переменной, используемой для доступа к локальному блоку данных потока
`0Ch`	`DWORD`	CallBack Table VA	Виртуальный адрес таблицы обратных вызовов. Локальные обратные вызовы — массив виртуальных адресов функций, которые будут вызваны загрузчиком после создания потока (нити, цепочки) и после его завершения. Последний вход имеет нулевое значение и указывает на конец таблицы

Секция ресурсов (*.rdata*)

Ресурсы представляют собой многоуровневое двоичное дерево. Их структура позволяет содержать до 2^{31} уровней, однако реально используются только три: верхний — Type, затем Name и затем Language (тип, имя, язык). Перемещения по иерархии каталогов ресурсов похожи на перемещения по каталогам жесткого диска.

Секция ресурсов в загрузочном модуле представлена следующими структурами данных:

- каталог ресурсов (Resources Directory Table);
- данные ресурсов (Resources Data).

Структура каталога ресурсов показана в табл. П4.9.

Таблица П4.9. *Каталог ресурсов*

Смещение	**Длина поля**	**Название поля**	**Описание поля**
`00h`	`DWORD`	Flags	Поле зарезервировано, должно быть равно нулю
`04h`	`DWORD`	Time/Date Stamp	Дата и время создания ресурсов компилятором ресурсов
`08h`	`WORD`	Major Version	Старшая часть номера версии ресурсов. Обычно равна нулю
`0Ah`	`WORD`	Minor Version	Младшая часть номера версии ресурсов. Обычно равна нулю
`0Ch`	`WORD`	Name Entry	Количество входов в таблицу имен (элементов массива) ресурсов. Таблица располагается в самом начале массива входов и содержит строковые имена, ассоциируемые с ресурсами
`0Eh`	`WORD`	ID_Num Entry	Количество 32-битных идентификаторов ресурсов

Сразу за каталогом ресурсов следует массив переменной длины, содержащий ресурсные входы. Поле Name Entry содержит число ресурсных входов, имеющих имена (связанные с каждым входом). Имена нечувствительны к регистру и расположены в порядке возрастания. Поле ID_Num Entry определяет число входов, имеющих в качестве имени 32-битовый идентификатор. Эти входы также отсортированы по возрастанию. Такая структура позволяет получать быстрый доступ к ресурсам по имени или по идентификатору,

но для отдельно взятого ресурса поддерживается только один вариант поиска (что согласуется с синтаксисом RC- и RES-файлов). Формат входа в таблицу ресурсов показан в табл. П4.10.

Таблица П4.10. *Вход в таблицу ресурсов*

Смещение	**Длина поля**	**Название поля**	**Описание поля**
`00h`	`DWORD`	Name RVA or Res ID	Поле содержит либо идентификатор ресурса, либо указатель на его имя в таблице имен ресурсов
`04h`	`DWORD`	Data Entry RVA or SubDirectory RVA	Указывает либо на данные, либо на еще одну таблицу входов ресурсов. Старший бит поля, сброшенный в ноль, говорит, что поле указывает на данные

Каждый элемент данных (Resource Entry Item) имеет формат, представленный в табл. П4.11.

Таблица П4.11. *Элемент данных ресурса*

Смещение	**Длина поля**	**Название поля**	**Описание поля**
`00h`	`DWORD`	Data RVA	Указатель на реально расположенные данные относительно Image Base
`04h`	`DWORD`	Size	Размер ресурсных данных
`08h`	`DWORD`	CodePage	Кодовая страница
`0Ch`	`DWORD`	Reserved	Не используется и устанавливается в ноль

Таблица настроек адресов

Если исполняемый файл не может быть загружен по адресу, который указал компоновщик, то загрузчик производит настройку модуля, используя данные из секции `.reloc`. Поправки задают смещения тех элементов загрузочного модуля, к которым следует прибавить некоторую величину.

Формирование данных поправок выглядит следующим образом. Поправки упаковываются сериями смежных фрагментов различной длины. Каждый фрагмент описывает поправки для одной четырехбайтовой страницы загрузочного модуля; структура фрагмента приведена в табл. П4.12.

Таблица П4.12. Фрагмент таблицы настроек

Смещение	Длина поля	Название поля	Описание поля
00h	DWORD	Page RVA	Относительный адрес страницы
04h	DWORD	Block Size	Размер блока настроек (с заголовком). Эта величина используется для вычисления количества настроек
08h	WORD	TypeOffset Record	Массив записей настроек, их переменное количество

Чтобы выполнить настройку, необходимо вычислить 32-битную разницу ("дельта") между желаемой базой загрузки и действительной. Если образ программы загружен по требуемому адресу, то эта разница равна нулю и никакой настройки не требуется. Каждый блок настроек должен начинаться на DWORD-границе, для выравнивания блока можно пользоваться нулями. При настройке необходимую позицию в блоке вычисляют как сумму относительного адреса страницы и базового адреса загруженной программы.

Элемент массива настроек содержит следующие битовые поля:

- Type (биты 12—15) — тип настройки;
- Offset (биты 0—11) — смещение внутри 4-килобайтной страницы.

Возможные типы поправок и задаваемые ими действия приведены в табл. П4.13.

Таблица П4.13. Типы поправок

Значение Type	Действие
0h	Адрес абсолютный, и никаких изменений производить не требуется
1h	Добавить старшие 16 битов "дельты" к 16-битному полю, находящемуся по смещению Offset. 16-битовое поле представляет старшие биты 32-битного слова
2h	Добавить младшие 16 битов "дельты" по смещению Offset. 16-битное поле представляет младшую половину 32-битного слова. Данная запись настройки присутствует только на RISC-машине, когда Object align равно по умолчанию 64 Кбайт
3h	Прибавляет 32-битное "дельта" к 32-битному значению

Таблица П4.13 *(окончание)*

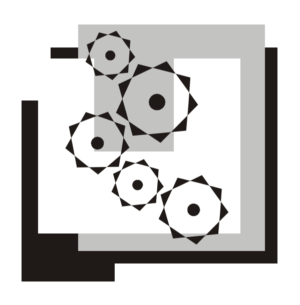
Значение Type	Действие
4h	Настройка требует полного 32-битного значения. Старшие 16 битов берутся по адресу Offset, а младшие — в следующем элементе TypeOffset. Все это объединяется в знаковую переменную, затем добавляется 32-битное "дельта" и DWORD 8000h. Старшие 16 битов получившегося значения сохраняются по адресу Offset в 16-битном поле
5h	Данная поправка предназначена для микропроцессоров с архитектурой, отличной от архитектуры Intel

Отладочная информация (*.debug$S*, *.debug$T*)

Здесь помещается структура отладочного каталога, создаваемого любыми компоновщиками. Другая отладочная информация зависит от транслятора. Структуру отладочной информации в формате COFF можно посмотреть в [2]. Структура отладочного каталога приведена в табл. П4.14.

Таблица П4.14. *Отладочный каталог*

Смещение	Длина поля	Название поля	Описание поля
00h	DWORD	Debug Flags	Флаги, по-видимому, не используются и устанавливаются в нулевое значение
04h	DWORD	Time/Date Stamp	Дата и время создания отладочной информации
08h	WORD	Major Version	Старший номер версии отладочной информации
0Ah	WORD	Minor Version	Младший номер версии отладочной информации
0Ch	DWORD	Debug Type	Тип информации для отладчика: • 0000h — UNKNOWN/BORLAND; • 0001h — таблица символов в COFF-формате; • 0002h — таблица символов в формате CodeView; • 0003h — таблица символов в FPO-формате; • 0004h — MISC; • 0005h — EXCEPTION; • 0006h — FIXUP
10h	DWORD	Data Size	Размер в байтах данных для отладки (без учета заголовка)
14h	DWORD	Data RVA	Относительный адрес расположения отладочных данных в памяти
18h	DWORD	Data Seek	Смещение отладочных данных в файле



Приложение 5

Файл kern.inc, используемый в главе 4.6

```
;файл kern.inc, используемый в программах главы 4.6

PVOID                  typedef             PTR
PIRP                   typedef             PTR _IRP
NTSTATUS               typedef             DWORD
PKEVENT                typedef      PTR    KEVENT
PIO_STATUS_BLOCK       typedef      PTR    IO_STATUS_BLOCK
BOOLEAN typedef         BYTE
PCHAR                  typedef      PTR    BYTE
PWSTR                  typedef      PTR    WORD
KPROCESSOR_MODE        typedef             BYTE
CHAR                   typedef             BYTE
WCHAR                  typedef             WORD
DEVICE_TYPE            typedef             DWORD
IO_TYPE_DEVICE_QUEUE   equ          14h
KSPIN_LOCK             typedef             DWORD
IO_TYPE_DPC            equ          13h
PDWORD                 typedef      PTR    DWORD
PSECURITY_DESCRIPTOR   typedef      PTR
MAXIMUM_VOLUME_LABEL_LENGTH  equ   (32 * sizeof(WCHAR))

;константы, определяющие тип запроса
IRP_MJ_CREATE                       equ 0
IRP_MJ_CREATE_NAMED_PIPE            equ 1
IRP_MJ_CLOSE                        equ 2
IRP_MJ_READ                         equ 3
IRP_MJ_WRITE                        equ 4
IRP_MJ_QUERY_INFORMATION            equ 5
IRP_MJ_SET_INFORMATION              equ 6
IRP_MJ_QUERY_EA                     equ 7
IRP_MJ_SET_EA                       equ 8
```

```
IRP_MJ_FLUSH_BUFFERS                    equ 9
IRP_MJ_QUERY_VOLUME_INFORMATION         equ 0Ah
IRP_MJ_SET_VOLUME_INFORMATION           equ 0Bh
IRP_MJ_DIRECTORY_CONTROL                equ 0Ch
IRP_MJ_FILE_SYSTEM_CONTROL              equ 0Dh
IRP_MJ_DEVICE_CONTROL                   equ 0Eh
IRP_MJ_INTERNAL_DEVICE_CONTROL          equ 0Fh
IRP_MJ_SHUTDOWN                         equ 10h
IRP_MJ_LOCK_CONTROL                     equ 11h
IRP_MJ_CLEANUP                          equ 12h
IRP_MJ_CREATE_MAILSLOT                  equ 13h
IRP_MJ_QUERY_SECURITY                   equ 14h
IRP_MJ_SET_SECURITY                     equ 15h
IRP_MJ_POWER                            equ 16h
IRP_MJ_SYSTEM_CONTROL                   equ 17h
IRP_MJ_DEVICE_CHANGE                    equ 18h
IRP_MJ_QUERY_QUOTA                      equ 19h
IRP_MJ_SET_QUOTA                        equ 1Ah
IRP_MJ_PNP                              equ 1Bh
IRP_MJ_PNP_POWER                        equ IRP_MJ_PNP
IRP_MJ_MAXIMUM_FUNCTION                 equ 1Bh

VPB STRUCT
    fwType            WORD   IO_TYPE_VPB
    cbSize            WORD   ?
    Flags             WORD   ?
    VolumeLabelLength WORD   ?
    DeviceObject      PVOID  ?
    RealDevice        PVOID  ?
    SerialNumber      DWORD  ?
    ReferenceCount    DWORD  ?
    VolumeLabel       WORD   (MAXIMUM_VOLUME_LABEL_LENGTH / (sizeof WCHAR)) dup(?)
VPB ENDS
PVPB    typedef       PTR VPB

UNICODE_STRING STRUCT
       woLength             WORD   ?
       MaximumLength        WORD   ?
       Buffer               PWSTR  ?
UNICODE_STRING ENDS

SECTION_OBJECT_POINTERS STRUCT
DataSectionObject           PVOID  ?
       SharedCacheMap       PVOID  ?
       ImageSectionObject   PVOID  ?
```

```
SECTION_OBJECT_POINTERS ENDS

PSECTION_OBJECT_POINTERS typedef PTR SECTION_OBJECT_POINTERS

IO_COMPLETION_CONTEXT STRUCT
       Port   PVOID  ?
       Key    PVOID  ?
IO_COMPLETION_CONTEXT ENDS

PIO_COMPLETION_CONTEXT typedef PTR IO_COMPLETION_CONTEXT

LARGE_INTEGER UNION
    STRUCT
        LowPart    DWORD  ?
        HighPart   DWORD  ?
    ENDS
    STRUCT u
        LowPart    DWORD  ?
        HighPart   DWORD  ?
    ENDS
    QuadPart       QWORD ?
LARGE_INTEGER ENDS

LIST_ENTRY STRUCT
    Flink      PVOID  ?
    Blink      PVOID  ?
LIST_ENTRY ENDS

KDPC STRUCT
    woType              WORD          IO_TYPE_DPC
    Number              BYTE          ?
    Importance          BYTE          ?
    DpcListEntry        LIST_ENTRY    <>
    DeferredRoutine     PVOID         ?
    DeferredContext     PVOID         ?
    SystemArgument1     PVOID         ?
    SystemArgument2     PVOID         ?
    pdwLock             PDWORD        ?
KDPC ENDS

KDEVICE_QUEUE STRUCT
    fwType             WORD        IO_TYPE_DEVICE_QUEUE
    cbSize             WORD        ?
    DeviceListHead     LIST_ENTRY  <>
    ksLock             KSPIN_LOCK  ?
    Busy               BOOLEAN     ?
```

```
    db                  3           dup(?)
KDEVICE_QUEUE ENDS

KDEVICE_QUEUE_ENTRY STRUCT    ; sizeof = 10h
    DeviceListEntry   LIST_ENTRY  <>
    SortKey           DWORD       ?
    Inserted          BOOLEAN     ?
    Db                3           dup(?)
KDEVICE_QUEUE_ENTRY ENDS

WAIT_CONTEXT_BLOCK STRUCT
    WaitQueueEntry         KDEVICE_QUEUE_ENTRY   <>
    DeviceRoutine          PVOID   ?
    DeviceContext          PVOID   ?
    NumberOfMapRegisters   DWORD   ?
    DeviceObject           PVOID   ?
    CurrentIrp             PVOID   ?
    BufferChainingDpc      PVOID   ?
WAIT_CONTEXT_BLOCK ENDS

DISPATCHER_HEADER STRUCT
    byType          BYTE         ?
    Absolute        BYTE         ?
    cbSize          BYTE         ?
    Inserted        BYTE         ?
    SignalState     DWORD        ?
    WaitListHead    LIST_ENTRY   <>
DISPATCHER_HEADER ENDS

KEVENT STRUCT
    Header          DISPATCHER_HEADER   <>
KEVENT ENDS

FILE_OBJECT STRUCT
    fwType                 WORD      IO_TYPE_FILE
    cbSize                 WORD      ?
    DeviceObject           PVOID     ?
    Vpb                    PVOID     ?
    FsContext              PVOID     ?
    FsContext2             PVOID     ?
    SectionObjectPointer   PSECTION_OBJECT_POINTERS ?
    PrivateCacheMap        PVOID     ?
    FinalStatus            NTSTATUS  ?
    RelatedFileObject      PVOID     ?
    LockOperation          BOOLEAN   ?
    DeletePending          BOOLEAN   ?
```

```
    ReadAccess            BOOLEAN   ?
    WriteAccess           BOOLEAN   ?
    DeleteAccess          BOOLEAN   ?
    SharedRead            BOOLEAN   ?
    SharedWrite           BOOLEAN   ?
    SharedDelete          BOOLEAN   ?
    Flags                 DWORD     ?
    FileName              UNICODE_STRING   <>
    CurrentByteOffset     LARGE_INTEGER    <>
    Waiters               DWORD     ?
    Busy                  DWORD     ?
    LastLock              PVOID     ?
    kevLock               KEVENT    <>
    Event                 KEVENT    <>
    CompletionContext     PIO_COMPLETION_CONTEXT   ?
FILE_OBJECT ENDS

PFILE_OBJECT typedef PTR FILE_OBJECT

IO_STATUS_BLOCK STRUCT
    Status         NTSTATUS  ?
    Information    DWORD     ?
IO_STATUS_BLOCK ENDS

STATUS_DEVICE_CONFIGURATION_ERROR equ 00C0000182h
STATUS_SUCCESS equ 0

IO_STATUS_BLOCK STRUCT
    Status         NTSTATUS  ?
    Information    DWORD     ?
IO_STATUS_BLOCK ENDS

KAPC STRUCT
    fwType            WORD         IO_TYPE_APC
    cbSize            WORD         ?
    Spare0            DWORD        ?
    Thread            PVOID        ?
    ApcListEntry      LIST_ENTRY   <>
    KernelRoutine     PVOID        ?
    RundownRoutine    PVOID        ?
    NormalRoutine     PVOID        ?
    NormalContext     PVOID        ?
; следующие два поля должны быть вместе
    SystemArgument1   PVOID        ?
    SystemArgument2   PVOID        ?
    ApcStateIndex     CHAR         ?
```

```
    ApcMode             KPROCESSOR_MODE    ?
    Inserted            BOOLEAN      ?
    db              ?
KAPC ENDS

_IRP STRUCT
    fwType                WORD    ?
    cbSize                WORD    ?
    MdlAddress            PVOID   ?
    Flags                 DWORD   ?
    UNION AssociatedIrp
        MasterIrp         PVOID   ?
        IrpCount          DWORD   ?
        SystemBuffer      PVOID   ?
    ENDS

    ThreadListEntry    LIST_ENTRY       <>
    IoStatus           IO_STATUS_BLOCK  <>
    RequestorMode      BYTE    ?
    PendingReturned    BYTE    ?
    StackCount         BYTE    ?
    CurrentLocation    BYTE    ?
    Cancel             BYTE    ?
    CancelIrql         BYTE    ?
    ApcEnvironment     BYTE    ?
    AllocationFlags    BYTE    ?
    UserIosb           PIO_STATUS_BLOCK ?
    UserEvent          PKEVENT ?
    UNION Overlay
        STRUCT AsynchronousParameters
            UserApcRoutine    PVOID    ?
            UserApcContext    PVOID    ?
        ENDS
        AllocationSize     LARGE_INTEGER    <>
    ENDS

    CancelRoutine      PVOID   ?
    UserBuffer         PVOID   ?

    UNION Tail
        STRUCT Overlay
            UNION
                DeviceQueueEntry     KDEVICE_QUEUE_ENTRY   <>
                STRUCT
                    DriverContext    PVOID    4 dup(?)
                ENDS
```

```
            ENDS
            Thread                  PVOID    ?
            AuxiliaryBuffer         PCHAR    ?
            STRUCT
                ListEntry           LIST_ENTRY   <>
                UNION
                    PacketType      DWORD    ?
                ENDS
            ENDS
            OriginalFileObject      PFILE_OBJECT   ?
        ENDS

        Apc                         KAPC     <>
        CompletionKey               PVOID    ?
    ENDS
_IRP ENDS

DEVICE_OBJECT STRUCT
    fwType              WORD        IO_TYPE_DEVICE
    cbSize              WORD        ?
    ReferenceCount      DWORD       ?
    DriverObject        PVOID       ?
    NextDevice          PVOID       ?
    AttachedDevice      PVOID       ?
    PDEVICE_OBJECT
    CurrentIrp          PIRP        ?
    Timer               PVOID       ?
    Flags               DWORD       ?
    Characteristics     DWORD       ?
    Vpb                 PVPB        ?
    DeviceExtension     PVOID       ?
    DeviceType          DEVICE_TYPE ?
    StackSize           CHAR        ?
    db                  3           dup(?)
    UNION Queue
        ListEntry       LIST_ENTRY          <>
        Wcb             WAIT_CONTEXT_BLOCK  <>
    ENDS
    AlignmentRequirement    DWORD   ?
    DeviceQueue         KDEVICE_QUEUE   <>
    Dpc                 KDPC            <>
    ActiveThreadCount   DWORD       ?
    SecurityDescriptor PSECURITY_DESCRIPTOR ?
    DeviceLock          KEVENT      <>
    SectorSize          WORD        ?
    Spare1              WORD        ?
```

```
    DeviceObjectExtension   PVOID   ?
    PDEVOBJ_EXTENSION
    Reserved            PVOID       ?
DEVICE_OBJECT ENDS

PDEVICE_OBJECT    typedef  PTR DEVICE_OBJECT
PDRIVER_EXTENSION typedef  PTR DRIVER_EXTENSION
PUNICODE_STRING   typedef  PTR UNICODE_STRING

DRIVER_OBJECT STRUCT    ; sizeof= 0A8h
    fwType          WORD                IO_TYPE_DRIVER
    cbSize          WORD                ?
    DeviceObject    PDEVICE_OBJECT      ?
    Flags           DWORD               ?
    DriverStart     PVOID               ?
    DriverSize      DWORD               ?
    DriverSection   PVOID               ?
    DriverExtension PDRIVER_EXTENSION   ?
    DriverName      UNICODE_STRING      <>
    HardwareDatabase PUNICODE_STRING    ?
    FastIoDispatch  PVOID               ?
    DriverInit      PVOID               ?
    DriverStartIo   PVOID               ?
    DriverUnload    PVOID               ?
    MajorFunction   PVOID  (IRP_MJ_MAXIMUM_FUNCTION + 1) dup(?)
DRIVER_OBJECT ENDS

IO_STACK_LOCATION STRUCT
MajorFunction                   BYTE    ?
    MinorFunction               BYTE    ?
    Flags                       BYTE    ?
    Control                     BYTE    ?
    UNION Parameters
        STRUCT Create
            SecurityContext     PVOID   ?
            Options             DWORD   ?
            FileAttributes      WORD    ?
            ShareAccess         WORD    ?
            EaLength            DWORD   ?
        ENDS
        STRUCT Read
            dwLength            DWORD   ?
            Key                 DWORD   ?
            ByteOffset          LARGE_INTEGER    <>
        ENDS
        STRUCT Write
```

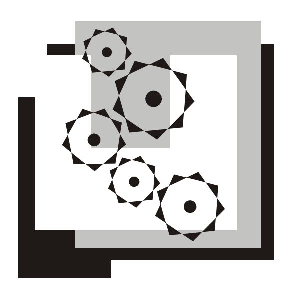
```
            dwLength                DWORD   ?
            Key                     DWORD   ?
            ByteOffset              LARGE_INTEGER    <>
        ENDS
        STRUCT QueryFile
            dwLength                DWORD   ?
            FileInformationClass    DWORD   ?
        ENDS

        STRUCT DeviceIoControl
            OutputBufferLength      DWORD   ?
            InputBufferLength       DWORD   ?
            IoControlCode           DWORD   ?
            Type3InputBuffer        PVOID   ?
        ENDS
        STRUCT ReadWriteConfig
            WhichSpace              DWORD   ?
            Buffer                  PVOID   ?
            dwOffset                DWORD   ?
            wdLength                DWORD   ?
        ENDS

        STRUCT SetLock
            bLock                   BOOLEAN  ?
            db 3 dup(?)
        ENDS

        STRUCT Others
            Argument1               PVOID   ?
            Argument2               PVOID   ?
            Argument3               PVOID   ?
            Argument4               PVOID   ?
        ENDS
    ENDS

    DeviceObject                    PDEVICE_OBJECT  ?
    FileObject                      PFILE_OBJECT    ?
    CompletionRoutine               PVOID           ?
    Context                         PVOID           ?
IO_STACK_LOCATION ENDS
```



Приложение 6

Пример консольного приложения с полной обработкой событий

В данном приложении я привожу полный текст консольного приложения, обрабатывающего события от клавиатуры и мыши. Эта программа является хорошей иллюстрацией к *главе 2.3*, посвященной консольным приложениям. Текст и исполняемый модуль имеется также на прилагаемом к книге компакт-диске.

```
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE           equ -11
STD_INPUT_HANDLE            equ -10
KEY_EVENT                   equ 1h
MENU_EVENT                  equ 8h
MOUSE_EVENT                 equ 2h
CAPSLOCK_ON                 equ 80h
ENHANCED_KEY                equ 100h
LEFT_ALT_PRESSED            equ 2h
LEFT_CTRL_PRESSED           equ 8h
NUMLOCK_ON                  equ 20h
RIGHT_ALT_PRESSED           equ 1h
RIGHT_CTRL_PRESSED          equ 4h
SCROLLLOCK_ON               equ 40h
SHIFT_PRESSED               equ 10h
DOUBLE_CLICK                equ 2h
MOUSE_MOVED                 equ 1h
MOUSE_WHEELED               equ 4h
FROM_LEFT_1ST_BUTTON_PRESSED equ 1h
RIGHTMOST_BUTTON_PRESSED    equ 2h
CTRL_C_EVENT                equ 0
CTRL_BREAK_EVENT            equ 1
```

```
CTRL_CLOSE_EVENT            equ 2
CTRL_LOGOFF_EVENT           equ 5
CTRL_SHUTDOWN_EVENT         equ 6

;прототипы функций API
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  FreeConsole@0:NEAR
EXTERN  AllocConsole@0:NEAR
EXTERN  ExitProcess@4:NEAR
EXTERN  CloseHandle@4:NEAR
EXTERN  ReadConsoleInputA@16:NEAR
EXTERN  wsprintfA:NEAR
EXTERN  Sleep@4:NEAR
EXTERN  SetConsoleCtrlHandler@8:NEAR
EXTERN  lstrlenA@4:NEAR
;структуры
uChar1 UNION
    UnicodeChar WORD ?
    AsciiChar   BYTE ?
uChar1 ENDS
KEY_EVENT_RECORD STRUC
    bKeyDown            DWORD   ?
    wRepeatCount        WORD    ?
    wVirtualKeyCode     WORD    ?
    wVirtualScanCode    WORD    ?
    uChar               uChar1  <0>
    dwControlKeyState   DWORD   ?
KEY_EVENT_RECORD ENDS
COORD1 STRUC
    X  WORD ?
    Y  WORD ?
COORD1 ENDS
MOUSE_EVENT_RECORD STRUC
    COORD               COORD1 <>
    dwButtonState       DWORD  ?
    dwControlKeyState   DWORD  ?
    dwEventFlags        DWORD  ?
MOUSE_EVENT_RECORD ENDS
WINDOW_BUFFER_SIZE_RECORD STRUC
    dwSize              COORD1 <>
WINDOW_BUFFER_SIZE_RECORD ENDS
MENU_EVENT_RECORD STRUC
    dwCommandId         DWORD  ?
MENU_EVENT_RECORD ENDS
FOCUS_EVENT_RECORD STRUC
```

```
    bSetFocus           DWORD  ?
FOCUS_EVENT_RECORD ENDS
Event1 UNION
    KeyEvent                 KEY_EVENT_RECORD          <>
    MouseEvent               MOUSE_EVENT_RECORD        <>
    WindowBufferSizeEvent   WINDOW_BUFFER_SIZE_RECORD <>
    MenuEvent                MENU_EVENT_RECORD         <>
    FocusEvent               FOCUS_EVENT_RECORD        <>
EVENT1 ENDS
INPUT_RECORD STRUC
    EventType           WORD  ?
                        DW    0   ;для выравнивания
    Event               Event1 <>
INPUT_RECORD ENDS
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;-----------------------------------------------
_DATA SEGMENT
    h1   DWORD  ?
    h2   DWORD  ?
    n    DWORD  ?
    i    DWORD  ?
    s1   DB "Error input!",13,10,0
    s2   DB 35 DUP(0)
    s4   DB "CTRL+C",13,10,0
    s5   DB "CTRL+BREAK",13,10,0
    s6   DB "CLOSE",13,10,0
    s7   DB "LOGOFF",13,10,0
    s8   DB "SHUTDOWN",13,10,0
    s9   DB "CTRL",13,10,0
    s10  DB "ALT",13,10,0
    s11  DB "SHIFT",13,10,0
    s12  DB " ",13,10,0
    s13  DB "Code %d ",13,10,0
    s14  DB "CAPSLOCK ",13,10,0
    s15  DB "NUMLOCK ",13,10,0
    s16  DB "SCROLLOCK ",13,10,0
    s17  DB "Enhanced key (virtual code) %d ",13,10,0
    s18  DB "Function key (virtual code) %d ",13,10,0
    s19  DB "Left mouse button",13,10,0
    s20  DB "Right mouse button",13,10,0
    s21  DB "Double click",13,10,0
    s22  DB "Wheel was rolled",13,10,0
    cc   WORD  ?
    fc   DB "Character '%c' ",13,10,0
```

```
    xy   DB "Location of cursor x=%d y=%d",13,10,0
    LENS DWORD ?
;массив из одного элемента
;теоретически массив должен быть, но не превышать
;объем памяти в 64 Кбайт, однако практически одного элемента хватает
    IR   INPUT_RECORD 1 DUP (<>)
    RES  DWORD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;инициализация консоли
    CALL FreeConsole@0
    CALL AllocConsole@0
;получить handle ввода
    PUSH STD_INPUT_HANDLE
    CALL GetStdHandle@4
    MOV  h2,EAX
;получить handle вывода
    PUSH STD_OUTPUT_HANDLE
    CALL GetStdHandle@4
    MOV  h1,EAX
;установить обработчик событий
    PUSH 1
    LEA  EAX,HANDLER
    PUSH EAX
    CALL SetConsoleCtrlHandler@8
;процедура с циклом обработки сообщений
    CALL INPUTCONS
;выход
    PUSH 0
    LEA  EAX,HANDLER
    PUSH EAX
    CALL SetConsoleCtrlHandler@8
    PUSH h1
    CALL CloseHandle@4
    PUSH h2
    CALL CloseHandle@4
    PUSH 0
    CALL ExitProcess@4
;обработчик критических консольных событий
;DWORD PTR [EBP+8] - единственный параметр обработчика
HANDLER PROC
    PUSH EBP
    MOV  EBP,ESP
    PUSH EBX
```

```
    PUSH ESI
    PUSH EDI
;параметр
    MOV  EAX,DWORD PTR [EBP+8]
;событие CTRL+C
    CMP  EAX,CTRL_C_EVENT
    JNZ  _N1
    PUSH OFFSET s4
    CALL PRINT
    JMP  _EXIT
_N1:
;событие CTRL+BREAK
    CMP  EAX,CTRL_BREAK_EVENT
    JNZ  _N2
    PUSH OFFSET s5
    CALL PRINT
    JMP  _EXIT
_N2:
;закрытие консоли
    CMP  EAX,CTRL_CLOSE_EVENT
    JNZ  _N3
    PUSH OFFSET s6
    CALL PRINT
    PUSH 2000
    CALL Sleep@4
    PUSH 0
    CALL ExitProcess@4
    JMP  _EXIT
_N3:
;завершение сеанса
    CMP  EAX,CTRL_LOGOFF_EVENT
    JNZ  _N4
    PUSH OFFSET s7
    CALL PRINT
    PUSH 2000
    CALL Sleep@4
    PUSH 0
    CALL ExitProcess@4
    JMP  _EXIT
_N4:
;завершение работы
    CMP  EAX,CTRL_SHUTDOWN_EVENT
    JNZ  _EXIT
    PUSH OFFSET s8
    CALL PRINT
    PUSH 2000
```

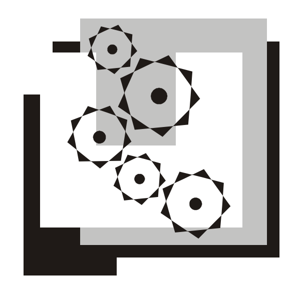
```
    CALL Sleep@4
    PUSH 0
    CALL ExitProcess@4
    JMP  _EXIT
_EXIT:
    POP  EDI
    POP  ESI
    POP  EBX
    MOV  ESP,EBP
    POP  EBP
;возвращаем ненулевое значение, указывая, что
;сами все обрабатываем
    MOV  EAX,1
    RET
HANDLER ENDP
;вывод на консоль строки, адрес которой помещен в стеке
PRINT   PROC
    PUSH EBP
    MOV  EBP,ESP
    PUSH EBX
    PUSH 0
    PUSH OFFSET LENS
;поместить адрес строки в EBX
    MOV  EBX,DWORD PTR [EBP+8]
;длина строки
    PUSH EBX
    CALL lstrlenA@4
    PUSH EAX
;вывод строки на консоль
;адрес строки
    PUSH EBX
    PUSH h1
    CALL WriteConsoleA@20
    POP  EBX
    MOV  ESP,EBP
    POP  EBP
    RET  4
PRINT   ENDP
;процедура обработки нажатия клавиш и событий от мыши
INPUTCONS PROC
LOO:
    PUSH OFFSET RES
    PUSH 1
    PUSH OFFSET IR
    PUSH h2
    CALL ReadConsoleInputA@16
```

```
;проверить правильность выполнения функции
    CMP  EAX,0
    JNZ  NO_ZER
;сообщение об ошибке
    PUSH OFFSET s1
    CALL PRINT
;задержка и выход
    PUSH 5000
    CALL Sleep@4
    RET
NO_ZER:
;события от клавиатуры или мыши?
    CMP ir.EventType,KEY_EVENT
    JNZ _MOUSE
;проверяем, нажата или отпущена клавиша
    CMP ir.Event.KeyEvent.bKeyDown,1
    JNZ LOO
_OK1:
;в начале управляющие клавиши
;клавиша <Caps Lock>
    CMP ir.Event.KeyEvent.dwControlKeyState,CAPSLOCK_ON
    JNZ _DAL1
    PUSH OFFSET s14
    CALL PRINT
    JMP _US_KEY
_DAL1:
;расширенная клавиатура
    CMP   ir.Event.KeyEvent.dwControlKeyState,ENHANCED_KEY
    JNZ   _DAL2
    MOVZX EAX,ir.Event.KeyEvent.wVirtualKeyCode
    PUSH  EAX
    PUSH  OFFSET s17
    PUSH  OFFSET s2
    CALL  wsprintfA
    ADD   ESP,12
    PUSH  OFFSET s2
    CALL  PRINT
    JMP   LOO
_DAL2:
;левая клавиша <Alt>
    CMP  ir.Event.KeyEvent.dwControlKeyState,LEFT_ALT_PRESSED
    JNZ  _DAL3
    PUSH OFFSET s10
    CALL PRINT
    JMP  _US_KEY
_DAL3:
;правая клавиша <Alt>
```

```
    CMP  ir.Event.KeyEvent.dwControlKeyState,RIGHT_ALT_PRESSED
    JNZ  _DAL4
    PUSH OFFSET s10
    CALL PRINT
    JMP  _US_KEY
_DAL4:
;левая клавиша <Ctrl>
    CMP  ir.Event.KeyEvent.dwControlKeyState,LEFT_CTRL_PRESSED
    JNZ  _DAL5
    PUSH OFFSET s9
    CALL PRINT
    JMP  _US_KEY
_DAL5:
;правая клавиша <Ctrl>
    CMP  ir.Event.KeyEvent.dwControlKeyState,RIGHT_CTRL_PRESSED
    JNZ  _DAL6
    PUSH OFFSET s9
    CALL PRINT
    JMP  _US_KEY
_DAL6:
;клавиша <Shift>
    CMP  ir.Event.KeyEvent.dwControlKeyState,SHIFT_PRESSED
    JNZ  _DAL7
    PUSH OFFSET s11
    CALL PRINT
    JMP  _US_KEY
_DAL7:
;клавиша <Num LOCK>
    CMP  ir.Event.KeyEvent.dwControlKeyState,NUMLOCK_ON
    JNZ  _DAL8
    PUSH OFFSET s15
    CALL PRINT
    JMP  _US_KEY
_DAL8:
;клавиша <Scroll Lock>
    CMP  ir.Event.KeyEvent.dwControlKeyState,SCROLLLOCK_ON
    JNZ  _US_KEY
    PUSH OFFSET s16
    CALL PRINT
;обычные клавиши
_US_KEY:
    XOR EAX,EAX
    MOV AL,ir.Event.KeyEvent.uChar.AsciiChar
    CMP AL,32
    JB  _BEL
;вывод символа
    PUSH EAX
```

```
    PUSH OFFSET fc
    PUSH OFFSET s2
    CALL wsprintfA
    ADD  ESP,12
    PUSH OFFSET s2
    CALL PRINT
    JMP  _MOUSE
_BEL:
    CMP  AL,0
    JNZ  _UPR
;назовем эти клавиши функциональными
;для них ASCII-код равен 0
    MOVZX EAX,ir.Event.KeyEvent.wVirtualKeyCode
    PUSH EAX
    PUSH OFFSET s18
    PUSH OFFSET s2
    CALL wsprintfA
    ADD  ESP,12
    PUSH OFFSET s2
    CALL PRINT
    JMP  _MOUSE
_UPR:
;клавиши с управляющими кодами (0<CODE<32)
    PUSH EAX
    PUSH OFFSET s13
    PUSH OFFSET s2
    CALL wsprintfA
    ADD  ESP,12
    PUSH OFFSET s2
    CALL PRINT
;события от мыши
_MOUSE:
    CMP  ir.EventType,MOUSE_EVENT
    JNZ  LOO
;здесь определяем, какое событие произошло
;двойной щелчок
    CMP  ir.Event.MouseEvent.dwEventFlags,DOUBLE_CLICK
    JNZ  _DALM1
    PUSH OFFSET s21
    CALL PRINT
_DALM1:
;передвижение курсора
    CMP   ir.Event.MouseEvent.dwEventFlags,MOUSE_MOVED
    JNZ   _DALM2
    MOVZX EAX,ir.Event.MouseEvent.COORD.Y
    PUSH  EAX
```

```
    MOVZX EAX,ir.Event.MouseEvent.COORD.X
    PUSH  EAX
    PUSH  OFFSET xy
    PUSH  OFFSET s2
    CALL  wsprintfA
    ADD   ESP,12
    PUSH  OFFSET s2
    CALL  PRINT
_DALM2:
;колесико мыши
    CMP  ir.Event.MouseEvent.dwEventFlags,MOUSE_WHEELED
    JNZ  _DALM3
    PUSH OFFSET s22
    CALL PRINT
_DALM3:
;левая кнопка мыши
    CMP  ir.Event.MouseEvent.dwButtonState,FROM_LEFT_1ST_BUTTON_PRESSED
    JNZ  _DALM4
    PUSH OFFSET s19
    CALL PRINT
_DALM4:
;правая кнопка мыши
    CMP  ir.Event.MouseEvent.dwButtonState,RIGHTMOST_BUTTON_PRESSED
    JNZ  _DALM5
    PUSH OFFSET s20
    CALL PRINT
_DALM5:
    JMP  LOO
    RET
INPUTCONS ENDP
_TEXT ENDS
END START
```



Приложение 7

Описание компакт-диска

К книге прилагается компакт-диск со всеми примерами, которые приводятся или упоминаются в тексте. Каждый пример помещен в каталог, имя которого совпадает с номером листинга, соответствующего данному примеру.

Примеры, представляющие полные программы, содержат:

- текст программы;
- файл пакетной трансляции, предполагается, что пакет MASM располагается в каталоге C:\MASM32;
- объектный файл;
- исполняемый файл. В случае, если у вас нет пакета MASM32, вы можете проверить работоспособность программы, просто запустив исполняемый модуль.

Кроме того, на компакт-диске располагаются и листинги примеров, не являющихся полными программами. Они имеют расширение txt.

Транслируя приведенные на компакт-диске программы, следует не забывать о командах `INCLUDELIB`. Эти команды подключают к программе статические библиотеки. Предполагается, что стандартные библиотеки импорта располагаются в каталоге c:\masm32\lib\.

Все приведенные примеры были протестированы в операционных системах Windows XP, Windows Server 2003, Windows Vista.

Список литературы

Источники, отмеченные (*), я использовал в электронном виде.

1. Пирогов В. Ю. Assembler. Учебный курс. — М.: Нолидж, 2001.
2. Питрек Мэтт. Секреты системного программирования в Windows 95. — К.: Диалектика, 1996.
3. Гук М. Процессоры Intel от 8086 до Pentium II. — СПб.: Питер, 1998.
4. Шилдт Г. Программирование на С и С++ для Windows 95. — К.: БХВ-Киев, 1996.
5. Григорьев В. Л. Микропроцессор i486. Архитектура и программирование (в 4-х книгах). — М.: Энергоатомиздат, 1993.
6. Брамм П., Брамм Д. Микропроцессор 80386 и его программирование. — М.: Мир, 1990.
7. Kauler Barry. Windows Assembly Language and Systems Programming. (*)
8. Шагурин И. И., Бродин В. Б., Мозговой Г. П. Микропроцессор 80386. Описание и система команд. — М.: Малип, 1992.
9. Смит Б. Э., Джонсон М. Т. Архитектура и программирование микропроцессора INTEL 80386. — М.: Конкорд, 1992.
10. Морс С. П., Альберт Д. Д. Архитектура микропроцессора 80286. — М.: Радио и связь, 1990.
11. Страус Э. Микропроцессор 80286. — М.: Versus Ltd., 1992.
12. Петзольд Ч. Программирование для Windows. В двух томах. — СПб.: БХВ-Санкт-Петербург, 1997.
13. Зубков С. В. Assembler для DOS, Windows и UNIX. — М.: ДМК, 2000.
14. Дункан Р. Оптимизация программ на ассемблере. — Барнаул, 1993. (*)
15. Джефри Рихтер. WINDOWS. Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. — СПб.: Питер, 2001.

16. Джонсон М. Харт. Системное программирование в среде Win32. — М.: Вильямс, 2001.
17. Сорокина С., Тихонов А., Щербаков А. Программирование драйверов и систем безопасности. — СПб.: БХВ-Петербург, 2002.
18. Рихтер Дж., Кларк Дж. Программирование серверных приложений для Windows 2000. — СПб.: Питер, 2001.
19. Эпплман Д. Win32 API и Visual Basic. — СПб.: Питер, 2002.
20. Джонс Э., Оланд Д. Программирование в сетях Microsoft Windows. — СПб.: Питер, 2002.
21. Нанс Б. Программирование в локальных сетях. — Пермь: Издательство Пермского университета, 1992.
22. Пирогов В. Ю. Ассемблер для Windows (3-е издание). — СПб.: БХВ-Петербург, 2005.
23. Гук М., Юров В. Процессоры Pentium 4, Athlon и Duron. — СПб.: Питер, 2001.
24. Румянцев П. В. Азбука программирования в Win32 API. (*)
25. Румянцев П. В. Работа с файлами в Win32 API. (*)
26. Пирогов В. Ю. Ассемблер на примерах. — СПб.: БХВ-Петербург, 2005.
27. Пирогов В. Ю. Ассемблер и дизассемблирование. — СПб.: БХВ-Петербург, 2006.

Предметный указатель

3

3DNow 658

A, B

ANSI 25, 109
API-программирование 12
ASCII 109
Boot-сектор 284

C

COM 166
Conditional breakpoint on import 663
Conditional breakpoints 661
Conditional log breakpoint
on import 663
Conditional logging breakpoint 661

D

Device Context 119
DHCP 484
Direct3D 165
DirectDraw 165
DirectGraphics 165
DirectInput 165
DirectMusic 165
DirectPlay 165
DirectSound 165
DirectSound3D 165
DirectX 165
DNS 483
Dynamic Link Library 434

F

Fastcall 722
FAT32 277, 282
FPO-оптимизация 644

G

GDI 119
GDI+ 154
GDT 557, 791

H

Heap 563
HEX-редакторы 647
Hotkey 255

I

IDA Pro 650
IDT 558
INT 3 664
INVOKE 335
IRP 708, 765

K

Kernal 743
Kernel mode debugger 676

L

LDT 557, 820
LocalSystem 617

M

MAC 478
Mailslot 350
Memory for b-tree 652
Message breakpoint on ClassProc 662
MFT 285
MMX 792
  расширение 814
MSDN 155, 740
Mutex 433, 534

N, O

NTFS 277, 279, 285
Original Equipment Manufacture (OEM) 110, 193
OllyDbg:
  интерфейс 656
  исправление исполняемого модуля 665
  окно наблюдения 664
  поиск информации 665
  точка останова 660
OpenGL 177
Ordinary breakpoints 660

P

Parse files *(см. Файл разряженный)* 279
Pipes 344, 526

R

Register 591
Relocation table 645
Remote debugger 676
Reparse points 279
Resource 217
RISC 644

S

Service Control Manager (SCM) 617
Service Control Program (SCP) 615
Socket *(см. Сокет)* 485
SoftICE 676
  точка останова 684, 698
Stream 288
STRUC 332
Surface 166
Symbol Loader 676

T

TCP/IP 479
Thread 398
Toggle breakpoint on import 663
TSS 791, 821

U

UDP 480
Unicode 110, 111, 156
UNION 333

W, X, Z

W32Dasm 666
Windows Vista 154
XML 271
Z-порядок 86

А

Адрес:
  IP 481
    назначение 484
  loopback 482
  логический 819
  маскирование 482
  широковещательный 482
Адресация:
  линейная 43
  сегментная 555, 556
  страничная 555, 559
Акселератор 218, 243
Арифметический сопроцессор 605, 806

Ассемблер 11, 13
Атрибут 651
в NTFS 287
нерезидентный 287
файла 278, 288

Б, В

Буфер протокола окна команд 690
Ввод/вывод 361
асинхронный 362
синхронный 362
Взаимоисключение 433
Временные характеристики файлов 279
Всплывающие подсказки 384
Вытесняющая схема многозадачности 399

Г, Д

Горячая клавиша 255
Дамп 39
Дейтаграмма 480
Дескриптор 819
защиты (безопасности) 347
потока 412
приложения 404
процесса 404
сегмента 820
Дизассемблер 37
Динамическая библиотека 434, 673
создание 437
Динамическое связывание 155
Директива:
.CODE 31
.DATA 31
INCLUDE 21
INCLUDELIB 28
INVOKE 27
PROTO 28
макроассемблера 329
времени выполнения 340
Дисковое устройство 356
Домен 345, 350

З

Загрузочный сектор 284
Задание набора команд 339

И

Идентификатор:
потока 412
процесса 404
Имя файла 281

К

Канал 355
анонимный 355, 526
именованный 355
Класс приоритета 400
Команды микропроцессора 792—795
Комментарий 108
Консоль 191
Контекст:
процесса 683
устройства 119
Критическая секция 422
Куча 563

Л

Ловушка 572
Локальное время 280
Локальные переменные 301

М

Макроопределение 337
Макросы 691
Манифест 271
Маршрутизация 484
Меню 226
динамическое 243
Метка 329
в макроопределениях 338
Многозадачность, вытесняющая 399
Модель памяти, плоская 26, 744
Модуль, объектный 24, 25

Н, О

Немодальные диалоговые окна 235
Объединение 201, 333
Объект ядра 363
Окно:
  верхнего уровня 85
  диалоговое 218, 220
    модальное 221
  дочернее 85
  родительское 85
  собственное 85

П

Память:
  динамическая 563
  плоская 561, 744
  разделяемая 452
Переменная, локальная 58
Планировщик 398
Поверхность 166
Поиск:
  рекурсивный 301, 471
  файлов 292
Порт 487
Поток 398, 412
  взаимодействие с другим потоком 419
  вторичный 398
  дескриптор 412
  идентификатор 412
  первичный 398
  приоритет 399
Почтовый ящик 350
Приложение:
  консольное 107, 191
  многопоточное 418
Приоритет потока 399
Программа:
  DEWIN.EXE 650
  dumpbin.exe 643
  editbin.exe 641
  Hiew.exe 647
  OllyDbg 646
  SoftICE 676
  W32Dasm 646, 666
Процедура:
  вызов 335
  обратного вызова 43, 364
Процесс 397
  дескриптор 404
  идентификатор 404
  создание 401
Путь файла 281

Р

Регистры микропроцессора 787
Редактор связей 19, 98
Режим адресации:
  защищенный 556
  реальный 555
Ресурс 217
  локальной сети 463

С

Связывание:
  неявное 442
  позднее 435, 440
  по имени 436
  по порядковому номеру 436
  раннее 435
  явное 435
Сегментация, упрощенная 31
Селектор 557, 820
Семафор 420
Синхроимпульсы 749
Системное время 280
Служба *(см. Сервис)* 615
Событие 422
Соглашение о вызовах 581
Сокет 345, 485
Сообщение:
  WM_PAINT 119
  WM_TIMER 208
  операционной системы Windows 783
Список 263
Стек, структура 58
Строка 45, 332
Структура 332

Т

Таблица перемещений 645
Таймер в консольном приложении 208
Текстовый файл 311
Точка останова 660, 684
  аппаратная 664
  в окне Memory 663
  на область памяти 663
  на сообщение Windows 661
  на функции импорта 663
  обычная 660
  условная 661
    с записью в журнал 661
Точка:
  монтирования томов 291
  повторной обработки *(см. Reparse points)* 291
  подсоединения каталогов 291
Транслятор ресурсов 217

У

Удаленная отладка 676
Упрощенная сегментация 31
Условное ассемблирование 333, 334

Ф

Файл 277
  имя 281
  двоичный 310
  отображаемый 513
  поиск 292
  путь 281
  разряженный *(см. Parse files)* 279
  текстовый, структура 318
Файловая система 277
  FAT32 282
  NTFS 285
Фильтр 572
Функции API 769

Ц

Цикл обработки сообщений 47

Ш

Шлюз 558

Я

Явное связывание 155
Ядро 743